Contents

Notices and Disclaimers ................................................................. 26
Introducing the Intel® Math Kernel Library .................................... 31
Getting Help and Support ............................................................... 32
What’s New ................................................................................... 33
Notational Conventions ................................................................. 34

Chapter 1: Overview
Performance Enhancements ........................................................... 38
Parallelism .................................................................................... 39
C Datatypes Specific to Intel MKL ............................................... 39

Chapter 2: BLAS and Sparse BLAS Routines
BLAS Routines .............................................................................. 41
Naming Conventions for BLAS Routines ........................................ 41
C Interface Conventions for BLAS Routines ................................... 42
Matrix Storage Schemes for BLAS Routines ................................... 44
BLAS Level 1 Routines and Functions ............................................ 44
cblas_?asum ................................................................................ 45
cblas_?axpy ............................................................................... 46
cblas_?copy ............................................................................... 46
cblas_?dot ................................................................................ 47
cblas_?sdot ............................................................................. 48
cblas_?dotc ............................................................................. 49
cblas_?dotu ............................................................................. 49
cblas_?nrm2 ............................................................................ 50
cblas_?rot ............................................................................... 51
cblas_?rotn ............................................................................ 52
cblas_?rotm ............................................................................ 52
cblas_?rotmg .......................................................................... 53
cblas_?scal ............................................................................. 55
cblas_?swap ............................................................................ 55
cblas_i?amax .......................................................................... 56
cblas_i?amin .......................................................................... 57
cblas_?cabs1 .......................................................................... 57

BLAS Level 2 Routines ................................................................ 58
cblas_?gbmv ........................................................................... 59
cblas_?gemv ........................................................................... 61
cblas_?ger ............................................................................... 63
cblas_?gerc ........................................................................... 64
cblas_?geru ........................................................................... 66
cblas_?hbmv ........................................................................... 67
cblas_?hemv ........................................................................... 70
cblas_?her ............................................................................... 71
cblas_?her2 ........................................................................... 72
cblas_?hpmv ........................................................................... 74
cblas_?hpr ............................................................................... 75
cblas_?hpr2 ........................................................................... 77
cblas_?sbmv ........................................................................... 78
cblas_?spmv ........................................................................... 81
cblas_?spr ................................................................. 82
cblas_?spr2 ......................................................... 84
cblas_?symv ......................................................... 85
cblas_?syr ............................................................ 87
cblas_?syr2 ........................................................... 88
cblas_?tbmv .......................................................... 89
cblas_?tbsv ............................................................ 92
cblas_?tpmv .......................................................... 95
cblas_?tpsv ............................................................ 97
cblas_?trmv ........................................................... 98
cblas_?trsv ........................................................... 100
BLAS Level 3 Routines ....................................................... 101
cblas_?gemm ......................................................... 102
cblas_?hemm ......................................................... 105
cblas_?herk ............................................................ 107
cblas_?her2k .......................................................... 109
cblas_?symm ......................................................... 112
cblas_?syrk ............................................................. 114
cblas_?syr2k ........................................................... 116
cblas_?trmm ........................................................... 119
cblas_?trsm ........................................................... 121
Sparse BLAS Level 1 Routines ...................................................... 123
Vector Arguments ................................................................ 123
Naming Conventions for Sparse BLAS Routines ......................... 123
Routines and Data Types ....................................................... 123
BLAS Level 1 Routines That Can Work With Sparse Vectors .......... 124
cblas_?axpyi ........................................................... 124
cblas_?doti ............................................................... 125
cblas_?dotci ............................................................ 126
cblas_?dotui ............................................................ 127
cblas_?gthr ............................................................. 127
cblas_?gthrz ............................................................ 128
cblas_?roti ............................................................... 129
cblas_?sctr ............................................................... 130
Sparse BLAS Level 2 and Level 3 Routines .................................... 130
Naming Conventions in Sparse BLAS Level 2 and Level 3 .......... 131
Sparse Matrix Storage Formats for Sparse BLAS Routines ............. 132
Routines and Supported Operations ......................................... 132
Interface Consideration ..................................................... 133
Sparse BLAS Level 2 and Level 3 Routines .................................... 138
mkl_?csrgemv ....................................................... 141
mkl_?bsrgemv ....................................................... 142
mkl_?coogemv ....................................................... 144
mkl_?diagemv ....................................................... 145
mkl_?csrsymv ....................................................... 147
mkl_?bsrsymv ....................................................... 148
mkl_?coosymv ....................................................... 150
mkl_?diasymv ....................................................... 151
mkl_?csrtrsv ....................................................... 152
mkl_?bsrtrsv ....................................................... 154
mkl_?cootrsv ....................................................... 156
mkl_?diatrsv ....................................................... 158
mkl_cspblas_?csrgemv ........................................ 159
mkl_cspblas_?bsrgemv ........................................ 161
mkl_cspblas_?coogemv ........................................ 162
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_cspblas_?csrsymv</td>
<td>164</td>
</tr>
<tr>
<td>mkl_cspblas_?bsrsymv</td>
<td>165</td>
</tr>
<tr>
<td>mkl_cspblas_?coosymv</td>
<td>167</td>
</tr>
<tr>
<td>mkl_cspblas_?csrtrsv</td>
<td>168</td>
</tr>
<tr>
<td>mkl_cspblas_?bsrtrsv</td>
<td>170</td>
</tr>
<tr>
<td>mkl_cspblas_?cootrsrv</td>
<td>172</td>
</tr>
<tr>
<td>mkl_?csrmv</td>
<td>173</td>
</tr>
<tr>
<td>mkl_?bsrmv</td>
<td>175</td>
</tr>
<tr>
<td>mkl_?cscmv</td>
<td>177</td>
</tr>
<tr>
<td>mkl_?coomv</td>
<td>179</td>
</tr>
<tr>
<td>mkl_?csrv</td>
<td>181</td>
</tr>
<tr>
<td>mkl_?bsrv</td>
<td>183</td>
</tr>
<tr>
<td>mkl_?cscsv</td>
<td>185</td>
</tr>
<tr>
<td>mkl_?coosv</td>
<td>187</td>
</tr>
<tr>
<td>mkl_?csrcmm</td>
<td>189</td>
</tr>
<tr>
<td>mkl_?bsrmm</td>
<td>191</td>
</tr>
<tr>
<td>mkl_?cscmm</td>
<td>194</td>
</tr>
<tr>
<td>mkl_?coomm</td>
<td>196</td>
</tr>
<tr>
<td>mkl_?csrcsm</td>
<td>199</td>
</tr>
<tr>
<td>mkl_?cscsm</td>
<td>201</td>
</tr>
<tr>
<td>mkl_?coosm</td>
<td>203</td>
</tr>
<tr>
<td>mkl_?bsrsm</td>
<td>205</td>
</tr>
<tr>
<td>mkl_?diamv</td>
<td>207</td>
</tr>
<tr>
<td>mkl_?skymv</td>
<td>209</td>
</tr>
<tr>
<td>mkl_?diasv</td>
<td>211</td>
</tr>
<tr>
<td>mkl_?skysv</td>
<td>213</td>
</tr>
<tr>
<td>mkl_?diamm</td>
<td>214</td>
</tr>
<tr>
<td>mkl_?skymm</td>
<td>217</td>
</tr>
<tr>
<td>mkl_?diasm</td>
<td>219</td>
</tr>
<tr>
<td>mkl_?skysm</td>
<td>221</td>
</tr>
<tr>
<td>mkl_?dnscsr</td>
<td>222</td>
</tr>
<tr>
<td>mkl_?csrcsr</td>
<td>225</td>
</tr>
<tr>
<td>mkl_?csrcoo</td>
<td>227</td>
</tr>
<tr>
<td>mkl_?csrbsr</td>
<td>229</td>
</tr>
<tr>
<td>mkl_?csrcsc</td>
<td>231</td>
</tr>
<tr>
<td>mkl_?csrcdia</td>
<td>233</td>
</tr>
<tr>
<td>mkl_?csrsky</td>
<td>235</td>
</tr>
<tr>
<td>mkl_?csradd</td>
<td>237</td>
</tr>
<tr>
<td>mkl_?csmultcsr</td>
<td>238</td>
</tr>
<tr>
<td>mkl_?csmultld</td>
<td>241</td>
</tr>
<tr>
<td>Sparse QR Routines</td>
<td>243</td>
</tr>
<tr>
<td>mkl_sparse_set_qr_hint</td>
<td>243</td>
</tr>
<tr>
<td>mkl_sparse_?_qr</td>
<td>244</td>
</tr>
<tr>
<td>mkl_sparse_qr_reorder</td>
<td>246</td>
</tr>
<tr>
<td>mkl_sparse_?_qr_factorize</td>
<td>247</td>
</tr>
<tr>
<td>mkl_sparse_?_qr.solve</td>
<td>248</td>
</tr>
<tr>
<td>mkl_sparse_?_qr_qmult</td>
<td>250</td>
</tr>
<tr>
<td>mkl_sparse_?_qr_rsolve</td>
<td>251</td>
</tr>
<tr>
<td>BLAS and LAPACK Compact Routines</td>
<td>253</td>
</tr>
<tr>
<td>mkl_?gemmcompact</td>
<td>257</td>
</tr>
<tr>
<td>mkl_?trsm_compact</td>
<td>260</td>
</tr>
<tr>
<td>mkl_?potrf_compact</td>
<td>262</td>
</tr>
<tr>
<td>mkl_?getrfnp_compact</td>
<td>263</td>
</tr>
<tr>
<td>mkl_?geqrf_compact</td>
<td>265</td>
</tr>
<tr>
<td>mkl_?getrinp_compact</td>
<td>266</td>
</tr>
</tbody>
</table>
Numerical Limitations for Compact BLAS and Compact LAPACK Routines ................................................................. 267
mkl_?get_size_compact .................................................................................. 268
mkl_get_format_compact .................................................................................. 269
mkl_?gepack_compact .................................................................................. 269
mkl_?geunpack_compact .................................................................................. 271
Inspector-executor Sparse BLAS Routines ................................................................. 272
Naming conventions in Inspector-executor Sparse BLAS Routines ........ 273
Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS Routines ................................................................. 274
Supported Inspector-executor Sparse BLAS Operations ...................... 274
Two-stage Algorithm for Inspector-executor Sparse BLAS routines ...... 275
Matrix Manipulation Routines .................................................................................. 276
  mkl_sparse_?_create_csr ............................................................................ 277
  mkl_sparse_?_create_csc ............................................................................ 278
  mkl_sparse_?_create_coo ............................................................................ 280
  mkl_sparse_?_create_bsr ............................................................................ 282
  mkl_sparse_copy ...................................................................................... 283
  mkl_sparse_destroy .................................................................................. 285
  mkl_sparse_convert_csr ............................................................................ 286
  mkl_sparse_convert_bsr ............................................................................ 287
  mkl_sparse_?_export_csr ............................................................................ 288
  mkl_sparse_?_export_csc ............................................................................ 289
  mkl_sparse_?_export_bsr ............................................................................ 291
  mkl_sparse_?_set_value ............................................................................ 293
  mkl_sparse_?_update_values ...................................................................... 293
  mkl_sparse_order ...................................................................................... 295
Inspector-executor Sparse BLAS Analysis Routines ................................................................. 295
  mkl_sparse_set_lu_smoother_hint .............................................................. 296
  mkl_sparse_set_mv_hint ............................................................................ 298
  mkl_sparse_set_sv_hint ............................................................................ 300
  mkl_sparse_set_mm_hint ............................................................................ 302
  mkl_sparse_set_sm_hint ............................................................................ 304
  mkl_sparse_set_dotmv_hint ...................................................................... 306
  mkl_sparse_set_symgs_hint ...................................................................... 307
  mkl_sparse_set_memory_hint ...................................................................... 309
  mkl_sparse_optimize .................................................................................. 310
Inspector-Executor Sparse BLAS Execution Routines ................................................................. 311
  mkl_sparse_?_lu_smoother ............................................................................ 312
  mkl_sparse_?_mv ...................................................................................... 315
  mkl_sparse_?_trsv ...................................................................................... 317
  mkl_sparse_?_mm ...................................................................................... 319
  mkl_sparse_?_trsm ...................................................................................... 322
  mkl_sparse_?_add ...................................................................................... 325
  mkl_sparse_spmm ...................................................................................... 327
  mkl_sparse_?_spmm ...................................................................................... 328
  mkl_sparse_sp2m ...................................................................................... 329
  mkl_sparse_sypr ....................................................................................... 333
  mkl_sparse_?_syprd ...................................................................................... 335
  mkl_sparse_?_symgs ...................................................................................... 338
  mkl_sparse_?_symgs_mv .................................................................................. 340
  mkl_sparse_syrk ....................................................................................... 342
  mkl_sparse_?_syrkd ...................................................................................... 343
  mkl_sparse_?_dotmv ................................................................................. 345
BLAS-like Extensions ...................................................................................... 348
Chapter 3: LAPACK Routines

Choosing a LAPACK Routine ................................................................. 398
C Interface Conventions for LAPACK Routines ..................................... 398
Matrix Layout for LAPACK Routines ..................................................... 400
Matrix Storage Schemes for LAPACK Routines .................................... 402
Mathematical Notation for LAPACK Routines ....................................... 409
Error Analysis ....................................................................................... 410
LAPACK Linear Equation Routines ....................................................... 411
LAPACK Linear Equation Computational Routines ............................... 411
Matrix Factorization: LAPACK Computational Routines ...................... 414
Solving Systems of Linear Equations: LAPACK Computational
Routines ................................................................................................. 459
Estimating the Condition Number: LAPACK Computational
Routines ................................................................................................. 499
Refining the Solution and Estimating Its Error: LAPACK
Computational Routines ........................................................................ 523
Matrix Inversion: LAPACK Computational Routines ............................. 578
Matrix Equilibration: LAPACK Computational Routines ................. 602
LAPACK Linear Equation Driver Routines ........................................... 618
?gesv .................................................................................. 619
?gesvx ................................................................................ 621
?gesvxx .............................................................. 626
?gbsv .............................................................. 633
?gbsvx ............................................................. 635
?gbsvxx ............................................................ 640
?gttsv .............................................................. 647
?gttsvx ......................................................................... 649
?dtsvb ........................................................................... 652
?posv .............................................................. 653
?posvx ........................................................................... 656
?posvxx ............................................................. 660
?ppsv .............................................................. 666
?ppsvx ......................................................................... 668
?pbsv .............................................................. 672
?pbsvx ......................................................................... 673
?ptsv ........................................................................... 677
<table>
<thead>
<tr>
<th>Routine</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>?ptsvx</td>
<td>678</td>
</tr>
<tr>
<td>?sysv</td>
<td>681</td>
</tr>
<tr>
<td>?sysv_aa</td>
<td>683</td>
</tr>
<tr>
<td>?sysv_rook</td>
<td>684</td>
</tr>
<tr>
<td>?sysv_rk</td>
<td>686</td>
</tr>
<tr>
<td>?sysvx</td>
<td>688</td>
</tr>
<tr>
<td>?sysvxx</td>
<td>691</td>
</tr>
<tr>
<td>?hesv</td>
<td>698</td>
</tr>
<tr>
<td>?hesv_aa</td>
<td>699</td>
</tr>
<tr>
<td>?hesv_rk</td>
<td>701</td>
</tr>
<tr>
<td>?hesvx</td>
<td>703</td>
</tr>
<tr>
<td>?hesvxx</td>
<td>706</td>
</tr>
<tr>
<td>?spsv</td>
<td>712</td>
</tr>
<tr>
<td>?spsvx</td>
<td>714</td>
</tr>
<tr>
<td>?hpsv</td>
<td>717</td>
</tr>
<tr>
<td>?hpsvx</td>
<td>718</td>
</tr>
</tbody>
</table>

LAPACK Least Squares and Eigenvalue Problem Routines ........................................... 721
LAPACK Least Squares and Eigenvalue Problem Computational Routines ....................... 722
Orthogonal Factorizations: LAPACK Computational Routines ........................................ 722
Singular Value Decomposition: LAPACK Computational Routines .................................... 777
Symmetric Eigenvalue Problems: LAPACK Computational Routines .................................. 795
Generalized Symmetric-Definite Eigenvalue Problems: LAPACK Computational Routines ........................................ 834
Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines .................................. 846
Generalized Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines .................................. 884
Generalized Singular Value Decomposition: LAPACK Computational Routines .................. 912
Cosine-Sine Decomposition: LAPACK Computational Routines ..................................... 928
LAPACK Least Squares and Eigenvalue Problem Driver Routines .................................... 935
Linear Least Squares (LLS) Problems: LAPACK Driver Routines .................................... 935
Generalized Linear Least Squares (LLS) Problems: LAPACK Driver Routines .................... 943
Symmetric Eigenvalue Problems: LAPACK Driver Routines ........................................ 947
Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines .................................... 994
Singular Value Decomposition: LAPACK Driver Routines ...................................... 1006
Cosine-Sine Decomposition: LAPACK Driver Routines ........................................ 1032
Generalized Symmetric Definite Eigenvalue Problems: LAPACK Driver Routines ................ 1037
Generalized Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines .................. 1076
LAPACK Auxiliary Routines ............................................................................................ 1100
?lacgv .................................................................................................................. 1100
?lacrm .................................................................................................................. 1100
?syconv .............................................................................................................. 1101
?syr ..................................................................................................................... 1103
imax1 .................................................................................................................. 1104
?sum1 .................................................................................................................. 1105
?gelq2 .................................................................................................................. 1105
?geqr2 .................................................................................................................. 1106
?geqrt2 ............................................................................................................... 1108
?geqrt3 ............................................................................................................... 1109
?getf2 ............................................................................................................... 1111
Chapter 4: ScaLAPACK Routines

Overview of ScaLAPACK Routines ..................................................1234
ScaLAPACK Array Descriptors .......................................................1235
Naming Conventions for ScaLAPACK Routines ...........................1237
ScaLAPACK Computational Routines ..........................................1238
Systems of Linear Equations: ScaLAPACK Computational Routines ....1238
Matrix Factorization: ScaLAPACK Computational Routines ..........1239
p?getrf ...............................................................................1239
p?gbtf2 ...............................................................................1241
p?dgbtf2 ..............................................................................1243
p?dttrf ...............................................................................1245
p?potrf ...............................................................................1247
p?pbtrf ...............................................................................1249
p?pttrf ...............................................................................1251
Solving Systems of Linear Equations: ScaLAPACK Computational Routines ..................................................1253
p?getrs ...............................................................................1253
p?gbtrs ...............................................................................1255
p?dgbtrs ..............................................................................1257
p?dttrs ...............................................................................1260
p?potrs ...............................................................................1262
p?pbtrs ...............................................................................1263
p?pttrs ...............................................................................1265
p?trtrs ...............................................................................1268
Estimating the Condition Number: ScaLAPACK Computational Routines1270
p?gecon ..............................................................................1270
p?pocon ...............................................................................1272
p?trcon ...............................................................................1275
Refining the Solution and Estimating Its Error: ScaLAPACK
Computational Routines ............................................................1277
p?gerfs ...............................................................................1277
p?porfs ...............................................................................1281
p?trrfs ...............................................................................1284
## Matrix Inversion: ScaLAPACK Computational Routines
- `p?getri` .................................................................................. 1287
- `p?potri` .................................................................................. 1289
- `p?trtri` .................................................................................. 1291

## Matrix Equilibration: ScaLAPACK Computational Routines
- `p?geequ` ............................................................................. 1292
- `p?poequ` ............................................................................. 1294

## Orthogonal Factorizations: ScaLAPACK Computational Routines
- `p?geqrf` ............................................................................... 1297
- `p?geqpf` .............................................................................. 1299
- `p?orgqr` ............................................................................... 1301
- `p?ungqr` ............................................................................... 1303
- `p?ormqr` ............................................................................... 1305
- `p?unmqr` ............................................................................... 1307
- `p?gelqf` ............................................................................... 1310
- `p?orglq` ............................................................................... 1312
- `p?unglq` ............................................................................... 1314
- `p?ormlq` ............................................................................... 1316
- `p?unmlq` ............................................................................... 1318
- `p?geqlf` ............................................................................... 1321
- `p?orgql` ............................................................................... 1323
- `p?ungql` ............................................................................... 1325
- `p?ormql` ............................................................................... 1327
- `p?unmql` ............................................................................... 1329
- `p?gerqf` ............................................................................... 1332
- `p?orgqr` ............................................................................... 1334
- `p?ungrq` ............................................................................... 1336
- `p?ormqr` ............................................................................... 1338
- `p?ormr3` ............................................................................... 1340
- `p?unmr3` ............................................................................... 1342
- `p?ormr` ............................................................................... 1344
- `p?unmr` ............................................................................... 1346
- `p?tzrzf` ............................................................................... 1349
- `p?ormrz` ............................................................................... 1352
- `p?unmrz` ............................................................................... 1354
- `p?ggqrf` ............................................................................... 1357
- `p?ggrqf` ............................................................................... 1361

## Symmetric Eigenvalue Problems: ScaLAPACK Computational Routines
- `p?syngst` ............................................................................ 1365
- `p?syntrd` ............................................................................ 1367
- `p?sytrd` ............................................................................... 1371
- `p?ormtr` ............................................................................... 1374
- `p?hengst` ............................................................................ 1377
- `p?hentrd` ............................................................................ 1379
- `p?unmtr` ............................................................................... 1383
- `p?stebz` ............................................................................... 1386
- `p?stedc` ............................................................................... 1389
- `p?stein` ............................................................................... 1392

## Nonsymmetric Eigenvalue Problems: ScaLAPACK Computational Routines
- `p?gehrd` ............................................................................. 1398
- `p?ormhr` ............................................................................. 1401
- `p?unmhr` ............................................................................. 1404
- `p?lahqr` ............................................................................. 1407
- `p?trevc` ............................................................................. 1410
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Singular Value Decomposition: ScaLAPACK Driver Routines</td>
<td>1412</td>
</tr>
<tr>
<td>p?gebrd</td>
<td>1413</td>
</tr>
<tr>
<td>p?ormbr</td>
<td>1416</td>
</tr>
<tr>
<td>p?unmbr</td>
<td>1420</td>
</tr>
<tr>
<td>Generalized Symmetric-Definite Eigenvalue Problems: ScaLAPACK</td>
<td></td>
</tr>
<tr>
<td>Computational Routines</td>
<td>1424</td>
</tr>
<tr>
<td>p?sygst</td>
<td>1424</td>
</tr>
<tr>
<td>p?hegst</td>
<td>1426</td>
</tr>
<tr>
<td>ScaLAPACK Driver Routines</td>
<td></td>
</tr>
<tr>
<td>p?gesv</td>
<td>1428</td>
</tr>
<tr>
<td>p?gesvx</td>
<td>1430</td>
</tr>
<tr>
<td>p?gbsv</td>
<td>1435</td>
</tr>
<tr>
<td>p?dbsv</td>
<td>1437</td>
</tr>
<tr>
<td>p?dtsv</td>
<td>1440</td>
</tr>
<tr>
<td>p?posv</td>
<td>1442</td>
</tr>
<tr>
<td>p?posvx</td>
<td>1444</td>
</tr>
<tr>
<td>p?pbsv</td>
<td>1449</td>
</tr>
<tr>
<td>p?ptsv</td>
<td>1451</td>
</tr>
<tr>
<td>p?gels</td>
<td>1453</td>
</tr>
<tr>
<td>p?syev</td>
<td>1456</td>
</tr>
<tr>
<td>p?syevd</td>
<td>1459</td>
</tr>
<tr>
<td>p?syevr</td>
<td>1462</td>
</tr>
<tr>
<td>p?syevx</td>
<td>1466</td>
</tr>
<tr>
<td>p?heev</td>
<td>1472</td>
</tr>
<tr>
<td>p?heevd</td>
<td>1475</td>
</tr>
<tr>
<td>p?heevr</td>
<td>1477</td>
</tr>
<tr>
<td>p?heevx</td>
<td>1482</td>
</tr>
<tr>
<td>p?gesvd</td>
<td>1489</td>
</tr>
<tr>
<td>p?sygsvx</td>
<td>1492</td>
</tr>
<tr>
<td>p?hegsvx</td>
<td>1499</td>
</tr>
<tr>
<td>ScaLAPACK Auxiliary Routines</td>
<td>1507</td>
</tr>
<tr>
<td>p?lacgv</td>
<td>1513</td>
</tr>
<tr>
<td>p?max1</td>
<td>1513</td>
</tr>
<tr>
<td>pilaver</td>
<td>1514</td>
</tr>
<tr>
<td>pmpcol</td>
<td>1515</td>
</tr>
<tr>
<td>pmpim2</td>
<td>1516</td>
</tr>
<tr>
<td>?combamax1</td>
<td>1517</td>
</tr>
<tr>
<td>p?sum1</td>
<td>1518</td>
</tr>
<tr>
<td>p?dbtrsv</td>
<td>1519</td>
</tr>
<tr>
<td>p?dttrsv</td>
<td>1521</td>
</tr>
<tr>
<td>p?gebal</td>
<td>1524</td>
</tr>
<tr>
<td>p?gebd2</td>
<td>1526</td>
</tr>
<tr>
<td>p?gehd2</td>
<td>1529</td>
</tr>
<tr>
<td>p?gelq2</td>
<td>1531</td>
</tr>
<tr>
<td>p?geql2</td>
<td>1533</td>
</tr>
<tr>
<td>p?geqr2</td>
<td>1535</td>
</tr>
<tr>
<td>p?gerq2</td>
<td>1537</td>
</tr>
<tr>
<td>p?getf2</td>
<td>1539</td>
</tr>
<tr>
<td>p?labrd</td>
<td>1541</td>
</tr>
<tr>
<td>p?lacon</td>
<td>1544</td>
</tr>
<tr>
<td>p?laconsb</td>
<td>1545</td>
</tr>
<tr>
<td>p?lacp2</td>
<td>1547</td>
</tr>
<tr>
<td>p?lacp3</td>
<td>1548</td>
</tr>
<tr>
<td>p?lacpy</td>
<td>1550</td>
</tr>
<tr>
<td>p?laevswp</td>
<td>1551</td>
</tr>
</tbody>
</table>
p?lahrd .................................................................1553
p?laicq ..............................................................1555
p?lamve ............................................................1556
p?lanh ..............................................................1557
p?lanhs ............................................................1559
p?lansy, p?lanhe ..............................................1561
p?lantr ..............................................................1563
p?lapiv ..............................................................1565
p?lapv2 ..............................................................1567
p?laqe ..............................................................1569
p?laqr0 ..............................................................1570
p?laqr1 ..............................................................1573
p?laqr2 ..............................................................1576
p?laqr3 ..............................................................1579
p?laqr5 ..............................................................1582
p?laqsy ..............................................................1584
p?lared1d ..........................................................1586
p?lared2d ..........................................................1587
p?larf ..............................................................1588
p?larfb ..............................................................1590
p?larfc ..............................................................1593
p?larfg ..............................................................1596
p?larft ..............................................................1598
p?larz ..............................................................1600
p?larzb ..............................................................1603
p?larzc ..............................................................1606
p?larzt ..............................................................1609
p?lascl ..............................................................1611
p?lase2 ..............................................................1613
p?laset ..............................................................1614
p?lasmsub ........................................................1616
p?lasrt ..............................................................1617
p?lassq ..............................................................1618
p?laswp ..............................................................1620
p?latra ..............................................................1622
p?latrd ..............................................................1623
p?latrs ..............................................................1626
p?latrz ..............................................................1628
p?lauu2 ..............................................................1630
p?lauum ...........................................................1631
p?lawil ..............................................................1633
p?org2l/p?ung2l ................................................1634
p?org2r/p?ung2r ................................................1636
p?orgl2/p?ungl2 ................................................1638
p?org2/p?ung2 ..................................................1640
p?orm2l/p?unm2l ............................................1642
p?orm2r/p?unm2r ............................................1645
p?orml2/p?unml2 .............................................1649
p?ormr2/p?unmr2 .............................................1652
p?pbtrsv ............................................................1655
p?pttrsv ............................................................1659
p?potf2 ............................................................1661
p?rot ...............................................................1663
p?rscl ................................................................1665
p?sygs2/p?hegs2 .............................................1666
Contents

Additional ScaLAPACK Routines ..............................................1739
ScaLAPACK Utility Functions and Routines .................................1740
ScaLAPACK Redistribution/Copy Routines .................................1746

Chapter 5: Sparse Solver Routines
Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface ..............1752
pardiso .................................................................1759
pardisoinit ..............................................................1765
pardiso_64 ...............................................................1766
pardiso_getenv, pardiso_setenv ..............................................1767
mkl_pardiso_pivot .......................................................1768
pardiso_getdiag ..........................................................1769
pardiso_export ............................................................1770
pardiso_handle_store .....................................................1772
pardiso_handle_restore ....................................................1773
Chapter 6: Extended Eigensolver Routines

The FEAST Algorithm ................................................................. 1869
Extended Eigensolver Functionality ....................................... 1870
Parallelism in Extended Eigensolver Routines .............................................. 1871
Achieving Performance With Extended Eigensolver Routines .................... 1871
Extended Eigensolver Interfaces for Eigenvalues within Interval .................. 1872
Extended Eigensolver Naming Conventions ............................................ 1872
feastinit ............................................................................................... 1873
Extended Eigensolver Input Parameters .................................................. 1874
Extended Eigensolver Output Details ..................................................... 1875
Extended Eigensolver RCI Routines ...................................................... 1876
Extended Eigensolver RCI Interface Description ..................................... 1876
?feast_srci/?feast_hrci .................................................................. 1879
Extended Eigensolver Predefined Interfaces ....................................... 1881
Matrix Storage .................................................................................. 1881
?feast_syev/?feast_heev .................................................................... 1882
?feast_sygv/?feast_hegv .................................................................... 1883
?feast_sbev/?feast_hbev .................................................................... 1885
?feast_sbgv/?feast_hbgv .................................................................... 1887
?feast_scvsrev/?feast_hcsrev .......................................................... 1889
?feast_scvsrgv/?feast_hcsrgv .......................................................... 1891

## Chapter 7: Vector Mathematical Functions

VM Data Types, Accuracy Modes, and Performance Tips .................................... 1894
VM Naming Conventions ........................................................................ 1895
VM Function Interfaces ........................................................................ 1896
VM Mathematical Function Interfaces ................................................. 1896
VM Pack Function Interfaces .............................................................. 1896
VM Unpack Function Interfaces .......................................................... 1896
VM Service Function Interfaces ........................................................... 1896
VM Input Function Interfaces ............................................................... 1896
VM Output Function Interfaces ............................................................ 1897
Vector Indexing Methods ...................................................................... 1897
VM Error Diagnostics .......................................................................... 1897
VM Mathematical Functions ................................................................ 1898
Special Value Notations ........................................................................ 1901
Arithmetic Functions ............................................................................ 1901
v?Add .............................................................................................. 1901
v?Sub .............................................................................................. 1903
v?Sqr .............................................................................................. 1904
v?Mul .............................................................................................. 1905
v?MulByConj ................................................................................. 1907
v?Conj ............................................................................................ 1908
v?Abs .............................................................................................. 1909
v?Arg .............................................................................................. 1910
v?LinearFrac .................................................................................. 1911
v?Fmod .......................................................................................... 1914
v?Remainder .................................................................................... 1915
Power and Root Functions ..................................................................... 1916
v?Inv .............................................................................................. 1916
v?Div .............................................................................................. 1917
v?Sqrt .............................................................................................. 1919
v?InvSqrt ....................................................................................... 1920
v?Cbrt .............................................................................................. 1921
v?InvCbrt ....................................................................................... 1922
v?Pow2o3 ...................................................................................... 1923
v?Pow3o2 ...................................................................................... 1924
v?Pow ............................................................................................ 1925
Exponential and Logarithmic Functions ............................................1932
v?Exp.................................................................................1932
v?Exp2 ...............................................................................1934
v?Exp10 .............................................................................1936
v?Expm1 ............................................................................1937
v?Ln...................................................................................1938
v?Log2 ...............................................................................1940
v?Log10 .............................................................................1941
v?Log1p .............................................................................1943
v?Logb ...............................................................................1944

Trigonometric Functions ................................................................1945
v?Cos.................................................................................1945
v?Sin .................................................................................1947
v?SinCos ............................................................................1948
v?CIS .................................................................................1950
v?Tan .................................................................................1951
v?Acos ...............................................................................1952
v?Asin ................................................................................1954
v?Atan ...............................................................................1956
v?Atan2 ...............................................................................1957
v?Cospi ..............................................................................1959
v?Sinpi ...............................................................................1960
v?Tanpi ...............................................................................1961
v?Acosp i .............................................................................1963
v?Asinpi ..............................................................................1964
v?Atanpi .............................................................................1965
v?Atan2pi ...........................................................................1966
v?Cosd ...............................................................................1968
v?Sind ................................................................................1969
v?Tand ...............................................................................1971

Hyperbolic Functions .............................................................1972
v?Cosh ...............................................................................1972
v?Sinh ...............................................................................1974
v?Tanh ...............................................................................1976
v?Acosh ..............................................................................1978
v?Asinh ..............................................................................1979
v?Atanh .............................................................................1981

Special Functions ......................................................................1983
v?Erf ..................................................................................1983
v?Erfc ..........................................................1985
v?CdfNorm ........................................................................1987
v?ErfInv .............................................................................1988
v?ErfcInv ...........................................................................1991
v?CdfNormInv .................................................................1992
v?LGamma ..............................................................1994
v?TGamma ..............................................................1995
v?ExpInt1 ...........................................................................1996

Rounding Functions .............................................................1997
v?Floor .................................................................1997
v?Ceil .................................................................1998
v?Trunc ..............................................................................1999
v?Round .............................................................................2000
Chapter 8: Statistical Functions

Random Number Generators .................................................................................................................. 2025
Random Number Generators Conventions .............................................................................................. 2026
  Random Number Generators Mathematical Notation .................................................................................. 2026
  Random Number Generators Naming Conventions ....................................................................................... 2027
Basic Generators ....................................................................................................................................... 2031
  BRNG Parameter Definition ...................................................................................................................... 2033
  Random Streams ......................................................................................................................................... 2034
  BRNG Data Types ........................................................................................................................................ 2034
Error Reporting ............................................................................................................................................. 2034
VS RNG Usage Model RNG Usage Model .................................................................................................... 2036
Service Routines .......................................................................................................................................... 2037
  vsNewStream ........................................................................................................................................... 2038
  vsNewStreamEx .......................................................................................................................................... 2040
  vsI1NewAbstractStream ............................................................................................................................. 2041
  vsI1NewAbstractStream ............................................................................................................................. 2042
  vsISNewAbstractStream ............................................................................................................................. 2044
  vsISNewI0AbstractStream ........................................................................................................................... 2045
  vsICopyStream .......................................................................................................................................... 2045
  vsICopyStreamState ................................................................................................................................... 2046
  vsISaveStreamF ......................................................................................................................................... 2047
  vsISaveStreamF ......................................................................................................................................... 2048
  vsISaveStreamM ......................................................................................................................................... 2049
  vsISaveStreamM ....................................................................................................................................... 2050
  vsIGetStreamSize ....................................................................................................................................... 2051
  vsISkipAheadStream ................................................................................................................................. 2052
  vsISkipAheadStreamEx .............................................................................................................................. 2054
  vsISkipAheadStreamEx .............................................................................................................................. 2056
  vsIGetStreamStateBrng ............................................................................................................................ 2057
null
Chapter 9: Fourier Transform Functions

FFT Functions ...........................................................................................................2191
FFT Interface .............................................................................................................2192
Computing an FFT ......................................................................................................2192
Configuration Settings ...............................................................................................2192
  DFTI_PRECISION ......................................................................................................2195
  DFTI_FORWARD_DOMAIN ..........................................................................................2195
  DFTI_DIMENSION, DFTI_LENGTHS ..........................................................................2196
  DFTI_PLACEMENT ......................................................................................................2196
  DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE ..................................................2197
  DFTI_NUMBER_OF_USER_THREADS .........................................................................2197
  DFTI_THREAD_LIMIT ..................................................................................................2197
  DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES ....................................................2198
  DFTI_NUMBER_OF_TRANSFORMS ..............................................................................2200
  DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE .................................................2200
  DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, 
  DFTI_CONJUGATE_EVEN_STORAGE ..........................................................................2201
  DFTI_PACKED_FORMAT .............................................................................................2204
  DFTI_WORKSPACE ....................................................................................................2213
  DFTI_COMMIT_STATUS ..............................................................................................2213
  DFTI_ORDERING ........................................................................................................2213
FFT Descriptor Manipulation Functions ........................................................................2214
  DftiCreateDescriptor ...............................................................................................2214
  DftiCommitDescriptor ...............................................................................................2215
  DftiFreeDescriptor ....................................................................................................2216
  DftiCopyDescriptor ....................................................................................................2217
FFT Descriptor Configuration Functions .........................................................................2218
  DftiSetValue ..............................................................................................................2218
  DftiGetValue ..............................................................................................................2219
FFT Computation Functions ........................................................................................2221
  DftiComputeForward .................................................................................................2221
  DftiComputeBackward ...............................................................................................2222
Configuring and Computing an FFT in C/C++ ...........................................................2224
Status Checking Functions ........................................................................................2227
  DftiErrorClass ..........................................................................................................2227
  DftiErrorMessage ....................................................................................................2228
Cluster FFT Functions .................................................................................................2230
Computing Cluster FFT ...............................................................................................2230
Distributing Data among Processes .............................................................................2230
Cluster FFT Interface .................................................................................................2232
Cluster FFT Descriptor Manipulation Functions .........................................................2233
  DftiCreateDescriptorDM .........................................................................................2233
  DftiCommitDescriptorDM .........................................................................................2234
  DftiFreeDescriptorDM ..............................................................................................2235
Cluster FFT Computation Functions ............................................................................2235
  DftiComputeForwardDM ............................................................................................2235
DftiComputeForwardDM ...............................................................................................2235
Chapter 10: PBLAS Routines

PBLAS Routines Overview ................................................................. 2243
PBLAS Routine Naming Conventions ................................................. 2244
PBLAS Level 1 Routines ................................................................. 2246
p?amax ...................................................................................... 2246
p?asum ...................................................................................... 2247
p?axpy ....................................................................................... 2248
p?copy ........................................................................................ 2250
p?dot ......................................................................................... 2251
p?dotc ........................................................................................ 2252
p?dotu ....................................................................................... 2253
p?nrm2 ....................................................................................... 2254
p?scal ......................................................................................... 2255
p?swap ....................................................................................... 2256
PBLAS Level 2 Routines ................................................................. 2258
p?gemv ...................................................................................... 2259
p?agemv ..................................................................................... 2261
p?ger ......................................................................................... 2263
p?gerc ........................................................................................ 2265
p?geru ....................................................................................... 2267
p?hemv ...................................................................................... 2268
p?ahemv ................................................................................... 2270
p?her ......................................................................................... 2272
p?her2 ....................................................................................... 2274
p?symv ....................................................................................... 2276
p?asymv ................................................................................... 2277
p?syr .......................................................................................... 2279
p?syr2 ....................................................................................... 2281
p?trmv ....................................................................................... 2283
p?atrmv ...................................................................................... 2285
p?trsm ....................................................................................... 2287
PBLAS Level 3 Routines ................................................................. 2289
p?geadd ...................................................................................... 2290
p?tradd ...................................................................................... 2291
p?gemm ...................................................................................... 2293
p?hemm ..................................................................................... 2295
p?herk ....................................................................................... 2297
p?her2k ...................................................................................... 2299
p?symm ...................................................................................... 2301
p?syrk ....................................................................................... 2304
p?syrk ....................................................................................... 2306
p?tran ......................................................................................... 2308
p?tranc ...................................................................................... 2309
p?trac ....................................................................................... 2310
p?trmm ...................................................................................... 2312
p?trsm ....................................................................................... 2314

Chapter 11: Partial Differential Equations Support

Trigonometric Transform Routines .................................................... 2317
Chapter 12: Nonlinear Optimization Problem Solvers

Nonlinear Solver Organization and Implementation .......................................................2368
Nonlinear Solver Routine Naming Conventions ..........................................................2369
Nonlinear Least Squares Problem without Constraints ................................................2370
  ?trnlsp_init ........................................................................2370
  ?trnlsp_check ......................................................................2372
  ?trnlsp_solve ......................................................................2373
  ?trnlsp_get ..........................................................................2375
  ?trnlsp_delete ......................................................................2376
Nonlinear Least Squares Problem with Linear (Bound) Constraints ..............................2377
  ?trnlspbc_init ......................................................................2377
  ?trnlspbc_check ..................................................................2379
  ?trnlspbc_solve ...................................................................2381
  ?trnlspbc_get ......................................................................2382
  ?trnlspbc_delete ..................................................................2383
Jacobian Matrix Calculation Routines ........................................................................2384
  ?jacobi_init ...........................................................................2384
  ?jacobi_solve ........................................................................2385
  ?jacobi_delete ......................................................................2386
  ?jacobi ................................................................................2387
  ?jacobix .............................................................................2388

Chapter 13: Support Functions
Chapter 14: BLACS Routines

Matrix Shapes......................................................................................2440
Repeatability and Coherence ..................................................................2441
BLACS Combine Operations ...................................................................2444
  ?gamx2d.....................................................................................2445
  ?gamm2d.....................................................................................2446
  ?gsum2d.....................................................................................2448
BLACS Point To Point Communication ......................................................2449
  ?gesd2d......................................................................................2451
  ?trsd2d.......................................................................................2452
  ?gerv2d......................................................................................2452
  ?trrv2d.......................................................................................2453
BLACS Broadcast Routines.....................................................................2454
  ?gebs2d......................................................................................2455
  ?trbs2d.......................................................................................2455
  ?gebr2d......................................................................................2456
  ?trbr2d.......................................................................................2457
BLACS Support Routines........................................................................2458
  Initialization Routines .................................................................2458
    blacs_pinfo .........................................................................2458
    blacs_setup.........................................................................2459
    blacs_get............................................................................2459
    blacs_set ............................................................................2461
    blacs_gridinit ......................................................................2462
    blacs_gridmap ......................................................................2463
  Destruction Routines .....................................................................2465
    blacs_freebuff .....................................................................2465
    blacs_gridexit .....................................................................2465
    blacs_abort ..........................................................................2466
    blacs_exit ...........................................................................2466
  Informational Routines ................................................................2467
    blacs_gridinfo ......................................................................2467
    blacs_pnum ..........................................................................2467
    blacs_pcoord ........................................................................2468
  Miscellaneous Routines .................................................................2468
    blacs_barrier ........................................................................2468
BLACS Routines Usage Examples............................................................2469

Chapter 15: Data Fitting Functions

Data Fitting Function Naming Conventions ............................................2470
Data Fitting Function Data Types ............................................................2471
Mathematical Conventions for Data Fitting Functions .................................2471
Data Fitting Usage Model ................................................................2474
Data Fitting Usage Examples .................................................................2474
Data Fitting Task Status and Error Reporting ........................................2480
Data Fitting Task Creation and Initialization Routines.................................2482
## Task Configuration Routines
- `df?EditPPSpline1D` (2484)
- `df?EditPtr` (2492)
- `dfiEditVal` (2493)
- `df?EditIdxPtr` (2495)
- `df?QueryPtr` (2497)
- `dfiQueryVal` (2498)
- `df?QueryIdxPtr` (2498)

## Data Fitting Computational Routines
- `df?Construct1D` (2500)
- `df?Interpolate1D/df?InterpolateEx1D` (2501)
- `df?Integrate1D/df?IntegrateEx1D` (2509)
- `df?SearchCells1D/df?SearchCellsEx1D` (2513)
- `df?InterpCallBack` (2515)
- `df?IntegrCallBack` (2516)
- `df?SearchCellsCallBack` (2518)

## Data Fitting Task Destructors
- `dfDeleteTask` (2520)

### Appendix A: Linear Solvers Basics
- Sparse Linear Systems... (2521)
- Matrix Fundamentals... (2521)
- Direct Method... (2522)
- Sparse Matrix Storage Formats... (2526)
- DSS Symmetric Matrix Storage... (2526)
- DSS Nonsymmetric Matrix Storage... (2527)
- DSS Structurally Symmetric Matrix Storage... (2528)
- DSS Distributed Symmetric Matrix Storage... (2529)
- Sparse BLAS CSR Matrix Storage Format... (2530)
- Sparse BLAS CSC Matrix Storage Format... (2532)
- Sparse BLAS Coordinate Matrix Storage Format... (2533)
- Sparse BLAS Diagonal Matrix Storage Format... (2534)
- Sparse BLAS Skyline Matrix Storage Format... (2534)
- Sparse BLAS BSR Matrix Storage Format... (2535)

### Appendix B: Routine and Function Arguments
- Vector Arguments in BLAS... (2539)
- Vector Arguments in VM... (2540)
- Matrix Arguments... (2541)

### Appendix C: FFTW Interface to Intel(R) Math Kernel Library
- Notational Conventions... (2545)
- FFTW2 Interface to Intel(R) Math Kernel Library... (2545)
- Wrappers Reference... (2545)
  - One-dimensional Complex-to-complex FFTs... (2545)
  - Multi-dimensional Complex-to-complex FFTs... (2546)
  - One-dimensional Real-to-half-complex/Half-complex-to-real FFTs... (2546)
  - Multi-dimensional Real-to-complex/Complex-to-real FFTs... (2546)
  - Multi-threaded FFTW... (2547)
  - FFTW Support Functions... (2547)
- Limitations of the FFTW2 Interface to Intel® MKL... (2548)
- Installing FFTW Interface Wrappers... (2548)
- Creating the Wrapper Library... (2548)
Notices and Disclaimers

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No product or component can be absolutely secure. Check with your system manufacturer or retailer or learn more at [intel.com].

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications. Current characterized errata are available on request.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Intel, the Intel logo, Intel Atom, Intel Core, Intel Xeon Phi, VTune and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Java is a registered trademark of Oracle and/or its affiliates.

Third Party Content

Intel® Math Kernel Library (Intel® MKL) includes content from several 3rd party sources that was originally governed by the licenses referenced below:

- Portions© Copyright 2001 Hewlett-Packard Development Company, L.P.
- Sections on the Linear Algebra PACKage (LAPACK) routines include derivative work portions that have been copyrighted:
- Intel® MKL supports LAPACK 3.5 set of computational, driver, auxiliary and utility routines under the following license:
  
  Copyright © 1992-2011 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

  Copyright © 2000-2011 The University of California Berkeley. All rights reserved.

  Copyright © 2006-2012 The University of Colorado Denver. All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
• Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
• Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
• Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The original versions of LAPACK from which that part of Intel® MKL was derived can be obtained from http://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

• The original versions of the Basic Linear Algebra Subprograms (BLAS) from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/blas/index.html.

• XBLAS is distributed under the following copyright:

Copyright © 2008-2009 The University of California Berkeley. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

• The original versions of the Basic Linear Algebra Communication Subprograms (BLACS) from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/blacs/index.html. The authors of BLACS are Jack Dongarra and R. Clint Whaley.

• The original versions of Scalable LAPACK (ScaLAPACK) from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/scalapack/index.html. The authors of ScaLAPACK are L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley.
• The original versions of the Parallel Basic Linear Algebra Subprograms (PBLAS) routines from which the respective part of Intel® MKL was derived can be obtained from http://www.netlib.org/scalapack/html/pblas_qref.html.

• PARDISO (PARallel DIrect SOlver)* in Intel® MKL was originally developed by the Department of Computer Science at the University of Basel (http://www.unibas.ch). It can be obtained at http://www.pardiso-project.org.

• The Extended Eigensolver functionality is based on the Feast solver package and is distributed under the following license:

Copyright © 2009, The Regents of the University of Massachusetts, Amherst.
Developed by E. Polizzi
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the University nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

• Some Fast Fourier Transform (FFT) functions in this release of Intel® MKL have been generated by the SPIRAL software generation system (http://www.spiral.net/) under license from Carnegie Mellon University. The authors of SPIRAL are Markus Puschel, Jose Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo.

• Open MPI is distributed under the New BSD license, listed below.

Most files in this release are marked with the copyrights of the organizations who have edited them. The copyrights below are in no particular order and generally reflect members of the Open MPI core team who have contributed code to this release. The copyrights for code used under license from other parties are included in the corresponding files.

Copyright © 2004-2010 The Trustees of Indiana University and Indiana University Research and Technology Corporation. All rights reserved.

Copyright © 2004-2010 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.

Copyright © 2004-2010 High Performance Computing Center Stuttgart, University of Stuttgart. All rights reserved.

Copyright © 2004-2008 The Regents of the University of California. All rights reserved.

Copyright © 2006-2010 Los Alamos National Security, LLC. All rights reserved.

Copyright © 2006-2010 Cisco Systems, Inc. All rights reserved.

Copyright © 2006-2010 Voltaire, Inc. All rights reserved.
Copyright © 2006-2011 Sandia National Laboratories. All rights reserved.
Copyright © 2006-2010 Sun Microsystems, Inc. All rights reserved. Use is subject to license terms.
Copyright © 2006-2010 The University of Houston. All rights reserved.
Copyright © 2006-2009 Myricom, Inc. All rights reserved.
Copyright © 2007-2008 UT-Battelle, LLC. All rights reserved.
Copyright © 2007-2010 IBM Corporation. All rights reserved.
Copyright © 1998-2005 Forschungszentrum Juelich, Juelich Supercomputing Centre, Federal Republic of Germany
Copyright © 2005-2008 ZIH, TU Dresden, Federal Republic of Germany
Copyright © 2007 Evergrid, Inc. All rights reserved.
Copyright © 2008 Chelsio, Inc. All rights reserved.
Copyright © 2008-2009 Institut National de Recherche en Informatique. All rights reserved.
Copyright © 2007 Lawrence Livermore National Security, LLC. All rights reserved.
Copyright © 2007-2009 Mellanox Technologies. All rights reserved.
Copyright © 2006-2010 QLogic Corporation. All rights reserved.
Copyright © 2008-2010 Oak Ridge National Labs. All rights reserved.
Copyright © 2006-2010 Oracle and/or its affiliates. All rights reserved.
Copyright © 2009 Bull SAS. All rights reserved.
Copyright © 2010 ARM Ltd. All rights reserved.
Copyright © 2010-2011 Alex Brick . All rights reserved.
Copyright © 2012 The University of Wisconsin-La Crosse. All rights reserved.
Copyright © 2013-2014 Intel, Inc. All rights reserved.
Copyright © 2011-2014 NVIDIA Corporation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

• The Safe C Library is distributed under the following copyright:

  Copyright (c)
  Permission is hereby granted, free of charge, to any person obtaining a copy of this software and
  associated documentation files (the "Software"), to deal in the Software without restriction, including
  without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
  of the Software, and to permit persons to whom the Software is furnished to do so, subject to the
  following conditions:

  The above copyright notice and this permission notice shall be included in all copies or substantial portions
  of the Software.

  THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
  INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
  PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
  LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT
  OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR
  OTHER DEALINGS IN THE SOFTWARE.

• HPL Copyright Notice and Licensing Terms

  Redistribution and use in source and binary forms, with or without modification, are permitted provided
  that the following conditions are met:

  1. Redistributions of source code must retain the above copyright notice, this list of conditions and the
     following disclaimer.
  2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the
     following disclaimer in the documentation and/or other materials provided with the distribution.
  3. All advertising materials mentioning features or use of this software must display the following
     acknowledgement: This product includes software developed at the University of Tennessee, Knoxville,
     Innovative Computing Laboratories.
  4. The name of the University, the name of the Laboratory, or the names of its contributors may not be
     used to endorse or promote products derived from this software without specific written permission.

© Intel Corporation.

This software and the related documents are Intel copyrighted materials, and your use of them is governed
by the express license under which they were provided to you (License). Unless the License provides
otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this software or the
related documents without Intel's prior written permission.

This software and the related documents are provided as is, with no express or implied warranties, other
than those that are expressly stated in the License.
Introducing the Intel® Math Kernel Library

The Intel® Math Kernel Library (Intel® MKL) improves performance with math routines for software applications that solve large computational problems. Intel® MKL provides BLAS and LAPACK linear algebra routines, fast Fourier transforms, vectorized math functions, random number generation functions, and other functionality.

NOTE
Intel® MKL provides limited input validation to minimize the performance overheads. It is your responsibility when using Intel® MKL to ensure that input data has the required format and does not contain invalid characters. These can cause unexpected behavior of the library. Examples of the inputs that may result in unexpected behavior:

* Not-a-number (NaN) and other special floating point values
* Large inputs may lead to accumulator overflow

As the Intel® MKL API accepts raw pointers, it is your application's responsibility to validate the buffer sizes before passing them to the library. The library requires subroutine and function parameters to be valid before being passed. While some Intel® MKL routines do limited checking of parameter errors, your application should check for NULL pointers, for example.

Intel® MKL is optimized for the latest Intel processors, including processors with multiple cores (see the Intel® MKL Release Notes for the full list of supported processors). Intel® MKL also performs well on non-Intel processors.

For more details about functionality provided by Intel® MKL, see the Function Domains section.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Getting Help and Support

Intel provides a support web site that contains a rich repository of self help information, including getting started tips, known product issues, product errata, license information, user forums, and more. Visit the Intel® MKL support website at http://www.intel.com/software/products/support/.
What's New

This Developer Reference documents Intel® Math Kernel Library (Intel® MKL) 2020 release for the C interface

NOTE
This publication, the Intel® Math Kernel Library Developer Reference, was previously known as the Intel® Math Kernel Library Reference Manual.

The manual has been updated to reflect enhancements to the product, besides improvements and error corrections.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Notational Conventions

This manual uses the following terms to refer to operating systems:

Windows* OS
This term refers to information that is valid on all supported Windows* operating systems.

Linux* OS
This term refers to information that is valid on all supported Linux* operating systems.

macOS*
This term refers to information that is valid on Intel®-based systems running the macOS* operating system.

This manual uses the following notational conventions:

- Routine name shorthand (for example, ?ungqr instead of cungqr/zungqr).
- Font conventions used for distinction between the text and the code.

Routine Name Shorthand

For shorthand, names that contain a question mark "?:" represent groups of routines with similar functionality. Each group typically consists of routines used with four basic data types: single-precision real, double-precision real, single-precision complex, and double-precision complex. The question mark is used to indicate any or all possible varieties of a function; for example:

\[ ?\text{swap} \]

Refers to all four data types of the vector-vector \( ?\text{swap} \) routine:
\[ \text{sswap}, \text{dswap}, \text{cswap}, \text{and zswap}. \]

Font Conventions

The following font conventions are used:

- **lowercase courier**
  - Code examples:
    - \[ a[k+i][j] = \text{matrix}[i][j]; \]
      - data types; for example, const float*

- **lowercase courier mixed with UpperCase courier**
  - Function names; for example, vmlSetMode

- **lowercase courier italic**
  - Variables in arguments and parameters description. For example, \( incx \).
  - Used as a multiplication symbol in code examples and equations and where required by the programming language syntax.
Function Domains

NOTE
Intel® MKL provides limited input validation to minimize the performance overheads. It is your responsibility when using Intel® MKL to ensure that input data has the required format and does not contain invalid characters. These can cause unexpected behavior of the library. Examples of the inputs that may result in unexpected behavior:

• Not-a-number (NaN) and other special floating point values
• Large inputs may lead to accumulator overflow

As the Intel® MKL API accepts raw pointers, it is your application's responsibility to validate the buffer sizes before passing them to the library. The library requires subroutine and function parameters to be valid before being passed. While some Intel® MKL routines do limited checking of parameter errors, your application should check for NULL pointers, for example.

The Intel® Math Kernel Library includes Fortran routines and functions optimized for Intel® processor-based computers running operating systems that support multiprocessing. In addition to the Fortran interface, Intel® MKL includes a C-language interface for the Discrete Fourier transform functions, as well as for the Vector Mathematics and Vector Statistics functions. For hardware and software requirements to use Intel® MKL, see Intel® MKL Release Notes.

NOTE
Functions calls at runtime for Intel® MKL libraries on the Microsoft Windows® operating system can utilize the function, LoadLibrary(), and related loading functions in static, dynamic, and single dynamic library linking models. These functions attempt to access the loader lock which when used within or at the same time as another DllMain function call, can lead to a deadlock. If possible, avoid making your calls to Intel® MKL in a DllMain function or at the same time as other calls to DllMain even on separate threads. Refer to Microsoft documentation about DllMain and Dynamic-Link Library Best Practices for more details.

BLAS Routines

The BLAS routines and functions are divided into the following groups according to the operations they perform:

• **BLAS Level 1 Routines** perform operations of both addition and reduction on vectors of data. Typical operations include scaling and dot products.
• **BLAS Level 2 Routines** perform matrix-vector operations, such as matrix-vector multiplication, rank-1 and rank-2 matrix updates, and solution of triangular systems.
• **BLAS Level 3 Routines** perform matrix-matrix operations, such as matrix-matrix multiplication, rank-k update, and solution of triangular systems.

Starting from release 8.0, Intel® MKL also supports the Fortran 95 interface to the BLAS routines.

Starting from release 10.1, a number of BLAS-like Extensions are added to enable the user to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations.
Sparse BLAS Routines
The Sparse BLAS Level 1 Routines and Functions and Sparse BLAS Level 2 and Level 3 Routines routines and functions operate on sparse vectors and matrices. These routines perform vector operations similar to the BLAS Level 1, 2, and 3 routines. The Sparse BLAS routines take advantage of vector and matrix sparsity: they allow you to store only non-zero elements of vectors and matrices. Intel® MKL also supports Fortran 95 interface to Sparse BLAS routines.

Sparse QR
Sparse QR in Intel® MKL is a set of routines used to solve sparse matrices with real coefficients and general structure. All Sparse QR routines can be divided into three steps: reordering, factorization, and solving. Currently, only CSR format is supported for the input matrix, and Sparse QR operates on the matrix handle used in all SpBLAS IE routines. (For details on how to create a matrix handle, refer to mkl-sparse-create-csr.)

LAPACK Routines
The Intel® Math Kernel Library fully supports the LAPACK 3.7 set of computational, driver, auxiliary and utility routines.

The original versions of LAPACK from which that part of Intel® MKL was derived can be obtained from http://www.netlib.org/lapack/index.html. The authors of LAPACK are E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen.

The LAPACK routines can be divided into the following groups according to the operations they perform:
- Routines for solving systems of linear equations, factoring and inverting matrices, and estimating condition numbers (see LAPACK Routines: Linear Equations).
- Routines for solving least squares problems, eigenvalue and singular value problems, and Sylvester's equations (see LAPACK Routines: Least Squares and Eigenvalue Problems).

Starting from release 8.0, Intel® MKL also supports the Fortran 95 interface to LAPACK computational and driver routines. This interface provides an opportunity for simplified calls of LAPACK routines with fewer required arguments.

Sparse Solver Routines
Direct sparse solver routines in Intel® MKL (see Chapter 8) solve symmetric and symmetrically-structured sparse matrices with real or complex coefficients. For symmetric matrices, these Intel® MKL subroutines can solve both positive-definite and indefinite systems. Intel® MKL includes a solver based on the PARDISO* sparse solver, referred to as Intel® MKL PARDISO, as well as an alternative set of user callable direct sparse solver routines.

If you use the Intel® MKL PARDISO sparse solver, please cite:

Intel® MKL provides also an iterative sparse solver (see Chapter 8) that uses Sparse BLAS level 2 and 3 routines and works with different sparse data formats.

Extended Eigensolver Routines
The Extended Eigensolver RCI Routines is a set of high-performance numerical routines for solving standard \((Ax = \lambda x)\) and generalized \((Ax = \mu Bx)\) eigenvalue problems, where \(A\) and \(B\) are symmetric or Hermitian. It yields all the eigenvalues and eigenvectors within a given search interval. It is based on the Feast algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which deviates fundamentally from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The Feast algorithm is inspired by the density-matrix representation and contour integration technique in quantum mechanics.
It is free from orthogonalization procedures. Its main computational tasks consist of solving very few inner independent linear systems with multiple right-hand sides and one reduced eigenvalue problem orders of magnitude smaller than the original one. The Feast algorithm combines simplicity and efficiency and offers many important capabilities for achieving high performance, robustness, accuracy, and scalability on parallel architectures. This algorithm is expected to significantly augment numerical performance in large-scale modern applications.

Some of the characteristics of the Feast algorithm [Polizzi09] are:

- Converges quickly in 2-3 iterations with very high accuracy
- Naturally captures all eigenvalue multiplicities
- No explicit orthogonalization procedure
- Can reuse the basis of pre-computed subspace as suitable initial guess for performing outer-refinement iterations

This capability can also be used for solving a series of eigenvalue problems that are close one another.

- The number of internal iterations is independent of the size of the system and the number of eigenpairs in the search interval
- The inner linear systems can be solved either iteratively (even with modest relative residual error) or directly

**VM Functions**

The Vector Mathematics functions (see Chapter 9) include a set of highly optimized implementations of certain computationally expensive core mathematical functions (power, trigonometric, exponential, hyperbolic, etc.) that operate on vectors of real and complex numbers.

Application programs that might significantly improve performance with VM include nonlinear programming software, integrals computation, and many others. VM provides interfaces both for Fortran and C languages.

**Statistical Functions**

Vector Statistics (VS) contains three sets of functions (see Chapter 10) providing:

- Pseudorandom, quasi-random, and non-deterministic random number generator subroutines implementing basic continuous and discrete distributions. To provide best performance, the VS subroutines use calls to highly optimized Basic Random Number Generators (BRNGs) and a set of vector mathematical functions.
- A wide variety of convolution and correlation operations.
- Initial statistical analysis of raw single and double precision multi-dimensional datasets.

**Fourier Transform Functions**

The Intel® MKL multidimensional Fast Fourier Transform (FFT) functions with mixed radix support (see Chapter 11) provide uniformity of discrete Fourier transform computation and combine functionality with ease of use. Both Fortran and C interface specification are given. There is also a cluster version of FFT functions, which runs on distributed-memory architectures and is provided only for Intel® 64 and Intel® Many Integrated Core architectures.

The FFT functions provide fast computation via the FFT algorithms for arbitrary lengths. See the Intel® MKL Developer Guide for the specific radices supported.

**Partial Differential Equations Support**

Intel® MKL provides tools for solving Partial Differential Equations (PDE) (see Chapter 13). These tools are Trigonometric Transform interface routines and Poisson Solver.

The Trigonometric Transform routines may be helpful to users who implement their own solvers similar to the Intel® MKL Poisson Solver. The users can improve performance of their solvers by using fast sine, cosine, and staggered cosine transforms implemented in the Trigonometric Transform interface.
The Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The Trigonometric Transform interface, which underlies the solver, is based on the Intel® MKL FFT interface (refer to Chapter 11), optimized for Intel® processors.

**Support Functions**

The Intel® MKL support functions (see Chapter 15) are used to support the operation of the Intel® MKL software and provide basic information on the library and library operation, such as the current library version, timing, setting and measuring of CPU frequency, error handling, and memory allocation.

Starting from release 10.0, the Intel® MKL support functions provide additional threading control.

Starting from release 10.1, Intel® MKL selectively supports a Progress Routine feature to track progress of a lengthy computation and/or interrupt the computation using a callback function mechanism. The user application can define a function called `mkl_progress` that is regularly called from the Intel® MKL routine supporting the progress routine feature. See the Progress Routines section in Chapter 15 for reference. Refer to a specific LAPACK or DSS/PARDISO function description to see whether the function supports this feature or not.

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

---

**Performance Enhancements**

The Intel® Math Kernel Library has been optimized by exploiting both processor and system features and capabilities. Special care has been given to those routines that most profit from cache-management techniques. These especially include matrix-matrix operation routines such as `dgemm()`.

In addition, code optimization techniques have been applied to minimize dependencies of scheduling integer and floating-point units on the results within the processor.

The major optimization techniques used throughout the library include:

- Loop unrolling to minimize loop management costs
- Blocking of data to improve data reuse opportunities
- Copying to reduce chances of data eviction from cache
- Data prefetching to help hide memory latency
- Multiple simultaneous operations (for example, dot products in `dgemm`) to eliminate stalls due to arithmetic unit pipelines
- Use of hardware features such as the SIMD arithmetic units, where appropriate

These are techniques from which the arithmetic code benefits the most.

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
Parallelism

In addition to the performance enhancements discussed above, Intel® MKL offers performance gains through parallelism provided by the symmetric multiprocessing performance (SMP) feature. You can obtain improvements from SMP in the following ways:

- One way is based on user-managed threads in the program and further distribution of the operations over the threads based on data decomposition, domain decomposition, control decomposition, or some other parallelizing technique. Each thread can use any of the Intel® MKL functions (except for the deprecated ?lacon LAPACK routine) because the library has been designed to be thread-safe.
- Another method is to use the FFT and BLAS level 3 routines. They have been parallelized and require no alterations of your application to gain the performance enhancements of multiprocessing. Performance using multiple processors on the level 3 BLAS shows excellent scaling. Since the threads are called and managed within the library, the application does not need to be recompiled thread-safe.
- Yet another method is to use tuned LAPACK routines. Currently these include the single- and double precision flavors of routines for QR factorization of general matrices, triangular factorization of general and symmetric positive-definite matrices, solving systems of equations with such matrices, as well as solving symmetric eigenvalue problems.

For instructions on setting the number of available processors for the BLAS level 3 and LAPACK routines, see Intel® MKL Developer Guide.

C Datatypes Specific to Intel MKL

The mkl_types.h file defines datatypes specific to Intel® MKL.

<table>
<thead>
<tr>
<th>C/C++ Type</th>
<th>Fortran Type</th>
<th>LP32 Equivalent (Size in Bytes)</th>
<th>LP64 Equivalent (Size in Bytes)</th>
<th>ILP64 Equivalent (Size in Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_INT</td>
<td>INTEGER</td>
<td>C/C++: int</td>
<td>C/C++: int</td>
<td>C/C++: long long</td>
</tr>
<tr>
<td>(MKL integer)</td>
<td>(default INTEGER)</td>
<td>Fortran: INTEGER*4 (4 bytes)</td>
<td>Fortran: INTEGER*4 (4 bytes)</td>
<td>(or define MKL_ILP64 macros)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Fortran: INTEGER*8 (8 bytes)</td>
</tr>
<tr>
<td>C/C++ Type</td>
<td>Fortran Type</td>
<td>LP32 Equivalent</td>
<td>LP64 Equivalent</td>
<td>ILP64 Equivalent</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------</td>
<td>------------------</td>
<td>------------------</td>
<td>------------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(Size in Bytes)</td>
<td>(Size in Bytes)</td>
<td>(Size in Bytes)</td>
</tr>
<tr>
<td>MKL_UINT</td>
<td>N/A</td>
<td>C/C++: unsigned int</td>
<td>C/C++: unsigned long int</td>
<td>C/C++: unsigned long int</td>
</tr>
<tr>
<td>(MKL unsigned integer)</td>
<td></td>
<td>(4 bytes)</td>
<td>(4 bytes)</td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>MKL_LONG</td>
<td>N/A</td>
<td>C/C++: long</td>
<td>C/C++: long</td>
<td>C/C++: long</td>
</tr>
<tr>
<td>(MKL long integer)</td>
<td></td>
<td>(4 bytes)</td>
<td>(Windows: 4 bytes)</td>
<td>(Linux, Mac: 8 bytes)</td>
</tr>
<tr>
<td>MKL_Complex8</td>
<td>COMPLEX*8</td>
<td>(8 bytes)</td>
<td>(8 bytes)</td>
<td>(8 bytes)</td>
</tr>
<tr>
<td>(Like C99 complex float)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MKL_Complex16</td>
<td>COMPLEX*16</td>
<td>(16 bytes)</td>
<td>(16 bytes)</td>
<td>(16 bytes)</td>
</tr>
<tr>
<td>(Like C99 complex double)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can redefine datatypes specific to Intel® MKL. One reason to do this is if you have your own types which are binary-compatible with Intel® MKL datatypes, with the same representation or memory layout. To redefine a datatype, use one of these methods:

- Insert the `#define` statement redefining the datatype before the `mkl.h` header file `#include` statement. For example,
  ```
  #define MKL_INT size_t
  #include "mkl.h"
  ```
- Use the compiler `-D` option to redefine the datatype. For example,
  ```
  ...-DMKL_INT=size_t...
  ```

**NOTE**
As the user, if you redefine Intel® MKL datatypes you are responsible for making sure that your definition is compatible with that of Intel® MKL. If not, it might cause unpredictable results or crash the application.
This chapter describes the Intel® Math Kernel Library implementation of the BLAS and Sparse BLAS routines, and BLAS-like extensions. The routine descriptions are arranged in several sections:

- **BLAS Level 1 Routines** (vector-vector operations)
- **BLAS Level 2 Routines** (matrix-vector operations)
- **BLAS Level 3 Routines** (matrix-matrix operations)
- **Sparse BLAS Level 1 Routines** (vector-vector operations).
- **Sparse BLAS Level 2 and Level 3 Routines** (matrix-vector and matrix-matrix operations)
- **BLAS-like Extensions**

The question mark in the group name corresponds to different character codes indicating the data type (s, d, c, and z or their combination); see *Routine Naming Conventions*.

When BLAS or Sparse BLAS routines encounter an error, they call the error reporting routine `xerbla`.

### BLAS Routines

#### Naming Conventions for BLAS Routines

BLAS routine names have the following structure:

```
<character> <name> <mod> ( )
```

The `<character>` field indicates the data type:

- `s` real, single precision
- `c` complex, single precision
- `d` real, double precision
- `z` complex, double precision

Some routines and functions can have combined character codes, such as `sc` or `dz`.

For example, the function `scasum` uses a complex input array and returns a real value.

The `<name>` field, in BLAS level 1, indicates the operation type. For example, the BLAS level 1 routines `?dot`, `?rot`, `?swap` compute a vector dot product, vector rotation, and vector swap, respectively.

In BLAS level 2 and 3, `<name>` reflects the matrix argument type:

- `ge` general matrix
- `gb` general band matrix
- `sy` symmetric matrix
- `sp` symmetric matrix (packed storage)
- `sb` symmetric band matrix
- `he` Hermitian matrix
- `hp` Hermitian matrix (packed storage)
- `hb` Hermitian band matrix
tr triangular matrix
tp triangular matrix (packed storage)
tb triangular band matrix.

The <mod> field, if present, provides additional details of the operation. BLAS level 1 names can have the following characters in the <mod> field:

c conjugated vector
u unconjugated vector
g Givens rotation construction
m modified Givens rotation
mg modified Givens rotation construction

BLAS level 2 names can have the following characters in the <mod> field:

mv matrix-vector product
sv solving a system of linear equations with a single unknown vector
r rank-1 update of a matrix
r2 rank-2 update of a matrix.

BLAS level 3 names can have the following characters in the <mod> field:

mm matrix-matrix product
sm solving a system of linear equations with multiple unknown vectors
rk rank-k update of a matrix
r2k rank-2k update of a matrix.

The examples below illustrate how to interpret BLAS routine names:

ddot <d> <dot>: double-precision real vector-vector dot product
cdotc <c> <dot> <c>: complex vector-vector dot product, conjugated
scasum <sc> <asum>: sum of magnitudes of vector elements, single precision real output and single precision complex input
cdotu <c> <dot> <u>: vector-vector dot product, unconjugated, complex
sgemv <s> <ge> <mv>: matrix-vector product, general matrix, single precision
ztrmm <z> <tr> <mm>: matrix-matrix product, triangular matrix, double-precision complex.

Sparse BLAS level 1 naming conventions are similar to those of BLAS level 1. For more information, see Naming Conventions.

C Interface Conventions for BLAS Routines

CBLAS, the C interface to the Basic Linear Algebra Subprograms (BLAS), provides a C language interface to BLAS routines for Intel® MKL. While you can call the Fortran implementation of BLAS, for coding in C the CBLAS interface has some advantages such as allowing you to specify column-major or row-major ordering with the layout parameter.
For more information about calling Fortran routines from C in general, and specifically about calling BLAS and CBLAS routines, see "Mixed-language Programming with the Intel® Math Kernel Library" in the Intel® Math Kernel Library Developer Guide.

NOTE
This reference contains syntax in C for both the CBLAS interface and the Fortran BLAS routines.

In CBLAS, the Fortran routine names are prefixed with cblas_ (for example, dasum becomes cblas_dasum). Names of all CBLAS functions are in lowercase letters.

Complex functions ?dotc and ?dotu become CBLAS subroutines (void functions); they return the complex result via a void pointer, added as the last parameter. CBLAS names of these functions are suffixed with _sub. For example, the BLAS function cblas_cdotc corresponds to cblas_cdotc_sub.

WARNING
Users of the CBLAS interface should be aware that the CBLAS are just a C interface to the BLAS, which is based on the FORTRAN standard and subject to the FORTRAN standard restrictions. In particular, the output parameters should not be referenced through more than one argument.

NOTE
This interface is not implemented in the Sparse BLAS Level 2 and Level 3 routines.

The arguments of CBLAS functions comply with the following rules:

- Input arguments are declared with the const modifier.
- Non-complex scalar input arguments are passed by value.
- Complex scalar input arguments are passed as void pointers.
- Array arguments are passed by address.
- BLAS character arguments are replaced by the appropriate enumerated type.
- Level 2 and Level 3 routines acquire an additional parameter of type CBLAS_LAYOUT as their first argument. This parameter specifies whether two-dimensional arrays are row-major (CblasRowMajor) or column-major (CblasColMajor).

Enumerated Types
The CBLAS interface uses the following enumerated types:

```c
enum CBLAS_LAYOUT {
    CblasRowMajor=101, /* row-major arrays */
    CblasColMajor=102}; /* column-major arrays */
enum CBLAS_TRANSPOSE {
    CblasNoTrans=111,   /* trans='N' */
    CblasTrans=112,     /* trans='T' */
    CblasConjTrans=113}; /* trans='C' */
enum CBLAS_UPLO {
    CblasUpper=121,    /* uplo ='U' */
    CblasLower=122};   /* uplo ='L' */
enum CBLAS_DIAG {
    CblasNonUnit=131, /* diag ='N' */
    CblasUnit=132};   /* diag ='U' */
enum CBLAS_SIDE {
    CblasLeft=141,     /* side ='L' */
    CblasRight=142};   /* side ='R' */
```
Matrix Storage Schemes for BLAS Routines

Matrix arguments of BLAS and CBLAS routines can use the following storage schemes:

- **Full storage**: a matrix \( A \) is stored in a two-dimensional array \( a \), with the matrix element \( A_{ij} \) stored in the array element \( a[i + j*lda] \) for column-major layout and \( a[j + i*lda] \) for row-major layout, where \( lda \) is the leading dimension for the array.

- **Packed storage** scheme allows you to store symmetric, Hermitian, or triangular matrices more compactly. For column-major layout, the upper or lower triangle of the matrix is packed by columns in a one dimensional array. For row-major layout, the upper or lower triangle of the matrix is packed by rows in a one dimensional array.

- **Band storage**: a band matrix is stored compactly in a two-dimensional array. For column-major layout, columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in a specific row of the array. For row-major layout, rows of the matrix are stored in the corresponding rows of the array, and diagonals of the matrix are stored in a specific column of the array.

For more information on matrix storage schemes, see Matrix Arguments in Appendix B.

Row-Major and Column-Major Layout

The BLAS routines follow the Fortran convention of storing two-dimensional arrays using column-major layout. When calling BLAS routines from C, remember that they require arrays to be in column-major format, not the row-major format that is the convention for C. Unless otherwise specified, the pseudo-code examples for the BLAS routines illustrate matrices stored using column-major layout.

The CBLAS interface allows you to specify either column-major or row-major layout for BLAS Level 2 and Level 3 routines, by setting the `layout` parameter to `CblasColMajor` or `CblasRowMajor`.

BLAS Level 1 Routines and Functions

BLAS Level 1 includes routines and functions, which perform vector-vector operations. The following table lists the BLAS Level 1 routine and function groups and the data types associated with them.

**BLAS Level 1 Routine and Function Groups and Their Data Types**

<table>
<thead>
<tr>
<th>Routine or Function Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cblas_?asum</code></td>
<td>s, d, sc, dz</td>
<td>Sum of vector magnitudes (functions)</td>
</tr>
<tr>
<td><code>cblas_?axpy</code></td>
<td>s, d, c, z</td>
<td>Scalar-vector product (routines)</td>
</tr>
<tr>
<td><code>cblas_?copy</code></td>
<td>s, d, c, z</td>
<td>Copy vector (routines)</td>
</tr>
<tr>
<td><code>cblas_?dot</code></td>
<td>s, d</td>
<td>Dot product (functions)</td>
</tr>
<tr>
<td><code>cblas_?ddot</code></td>
<td>sd, d</td>
<td>Dot product with double precision (functions)</td>
</tr>
<tr>
<td><code>cblas_?dotc</code></td>
<td>c, z</td>
<td>Dot product conjugated (functions)</td>
</tr>
<tr>
<td><code>cblas_?dotu</code></td>
<td>c, z</td>
<td>Dot product unconjugated (functions)</td>
</tr>
<tr>
<td><code>cblas_?nrm2</code></td>
<td>s, d, sc, dz</td>
<td>Vector 2-norm (Euclidean norm) (functions)</td>
</tr>
<tr>
<td><code>cblas_?rot</code></td>
<td>s, d, cs, zd</td>
<td>Plane rotation of points (routines)</td>
</tr>
<tr>
<td><code>cblas_?rotg</code></td>
<td>s, d, c, z</td>
<td>Generate Givens rotation of points (routines)</td>
</tr>
<tr>
<td><code>cblas_?rotm</code></td>
<td>s, d</td>
<td>Modified Givens plane rotation of points (routines)</td>
</tr>
<tr>
<td>Routine or Function Group</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>cblas_?rotmg</td>
<td>s, d</td>
<td>Generate modified Givens plane rotation of points (routines)</td>
</tr>
<tr>
<td>cblas_?scal</td>
<td>s, d, c, z, cs, zd</td>
<td>Vector-scalar product (routines)</td>
</tr>
<tr>
<td>cblas_?swap</td>
<td>s, d, c, z</td>
<td>Vector-vector swap (routines)</td>
</tr>
<tr>
<td>cblas_i?amax</td>
<td>s, d, c, z</td>
<td>Index of the maximum absolute value element of a vector (functions)</td>
</tr>
<tr>
<td>cblas_i?amin</td>
<td>s, d, c, z</td>
<td>Index of the minimum absolute value element of a vector (functions)</td>
</tr>
<tr>
<td>cblas_?cabs1</td>
<td>s, d</td>
<td>Auxiliary functions, compute the absolute value of a complex number of single or double precision</td>
</tr>
</tbody>
</table>

**cblas_?asum**

*Computes the sum of magnitudes of the vector elements.*

**Syntax**

```c
float cblas_sasum (const MKL_INT n, const float *x, const MKL_INT incx);
float cblas_scasum (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dasum (const MKL_INT n, const double *x, const MKL_INT incx);
double cblas_dzasum (const MKL_INT n, const void *x, const MKL_INT incx);
```

**Include Files**

- mkl.h

**Description**

The ?asum routine computes the sum of the magnitudes of elements of a real vector, or the sum of magnitudes of the real and imaginary parts of elements of a complex vector:

\[
\text{res} = |\text{Re}x_1| + |\text{Im}x_1| + |\text{Re}x_2| + |\text{Im}x_2| + \ldots + |\text{Re}x_n| + |\text{Im}x_n|,
\]

\[
\text{result} = \sum_{i=1}^{n} \left( |\text{Re}(X_i)| + |\text{Im}(X_i)| \right)
\]

where \( x \) is a vector with \( n \) elements.

**Input Parameters**

- \( n \)
  - Specifies the number of elements in vector \( x \).
- \( x \)
  - Array, size at least \((1 + (n-1)\times\text{abs}(\text{incx}))\).
- \( \text{incx} \)
  - Specifies the increment for indexing vector \( x \).

**Output Parameters**

- \( \text{res} \)
  - Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.
Return Values
Contains the sum of magnitudes of real and imaginary parts of all elements of the vector.

cblas_?axpy
Computes a vector-scalar product and adds the result to a vector.

Syntax
void cblas_saxpy (const MKL_INT n, const float a, const float *x, const MKL_INT incx, float *y, const MKL_INT incy);
void cblas_daxpy (const MKL_INT n, const double a, const double *x, const MKL_INT incx, double *y, const MKL_INT incy);
void cblas_caxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);
void cblas_zaxpy (const MKL_INT n, const void *a, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);

Include Files
• mkl.h

Description
The ?axpy routines perform a vector-vector operation defined as

\[ y := a \cdot x + y \]

where:
- \( a \) is a scalar
- \( x \) and \( y \) are vectors each with a number of elements that equals \( n \).

Input Parameters
- \( n \) Specifies the number of elements in vectors \( x \) and \( y \).
- \( a \) Specifies the scalar \( a \).
- \( x \) Array, size at least \((1 + (n-1) \cdot \text{abs}(\text{incx}))\).
- \( \text{incx} \) Specifies the increment for the elements of \( x \).
- \( y \) Array, size at least \((1 + (n-1) \cdot \text{abs}(\text{incy}))\).
- \( \text{incy} \) Specifies the increment for the elements of \( y \).

Output Parameters
- \( y \) Contains the updated vector \( y \).

cblas_?copy
Copies a vector to another vector.
### Syntax

```c
void cblas_scopy (const MKL_INT n, const float *x, const MKL_INT incx, float *y, const MKL_INT incy);
void cblas_dcopy (const MKL_INT n, const double *x, const MKL_INT incx, double *y, const MKL_INT incy);
void cblas_ccopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);
void cblas_zcopy (const MKL_INT n, const void *x, const MKL_INT incx, void *y, const MKL_INT incy);
```

### Include Files
- mkl.h

### Description

The `copy` routines perform a vector-vector operation defined as

\[ y = x, \]

where \( x \) and \( y \) are vectors.

#### Input Parameters

- **n**
  - Specifies the number of elements in vectors \( x \) and \( y \).
- **x**
  - Array, size at least \((1 + (n-1)\times\text{abs}(\text{incx}))\).
- **incx**
  - Specifies the increment for the elements of \( x \).
- **y**
  - Array, size at least \((1 + (n-1)\times\text{abs}(\text{incy}))\).
- **incy**
  - Specifies the increment for the elements of \( y \).

#### Output Parameters

- **y**
  - Contains a copy of the vector \( x \) if \( n \) is positive. Otherwise, parameters are unaltered.

### cblas_?dot

**Computes a vector-vector dot product.**

### Syntax

```c
float cblas_sdot (const MKL_INT n, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy);
double cblas_ddot (const MKL_INT n, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy);
```

### Include Files
- mkl.h

### Description

The `dot` routines perform a vector-vector reduction operation defined as
\[ \text{res} = \sum_{i=1}^{n} x_i \cdot y_i \]

where \( x_i \) and \( y_i \) are elements of vectors \( x \) and \( y \).

**Input Parameters**

- \( n \)  
  Specifies the number of elements in vectors \( x \) and \( y \).
- \( x \)  
  Array, size at least \((1+(n-1)\cdot\text{abs}(\text{incx}))\).
- \( \text{incx} \)  
  Specifies the increment for the elements of \( x \).
- \( y \)  
  Array, size at least \((1+(n-1)\cdot\text{abs}(\text{incy}))\).
- \( \text{incy} \)  
  Specifies the increment for the elements of \( y \).

**Return Values**

The result of the dot product of \( x \) and \( y \), if \( n \) is positive. Otherwise, returns 0.

**cblas_?sdot**

Computes a vector-vector dot product with double precision.

**Syntax**

```c
float cblas_sdsdot (const MKL_INT n, const float sb, const float *sx, const MKL_INT incx, const float *sy, const MKL_INT incy);
double cblas_dsdot (const MKL_INT n, const float *sx, const MKL_INT incx, const float *sy, const MKL_INT incy);
```
Output Parameters

res

Contains the result of the dot product of sx and sy (with sb added for sdsdot), if n is positive. Otherwise, res contains sb for sdsdot and 0 for dsdot.

Return Values

The result of the dot product of sx and sy (with sb added for sdsdot), if n is positive. Otherwise, returns sb for sdsdot and 0 for dsdot.

cblas_?dotc

Computes a dot product of a conjugated vector with another vector.

Syntax

void cblas_cdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotc);
void cblas_zdotc_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotc);

Include Files

• mkl.h

Description

The ?dotc routines perform a vector-vector operation defined as:

\[ res = \sum_{i=0}^{n} \text{conj}(x_i) \cdot y_i \]

where \( x_i \) and \( y_i \) are elements of vectors \( x \) and \( y \).

Input Parameters

n
Specifies the number of elements in vectors \( x \) and \( y \).

x
Array, size at least \((1 + (n -1) \cdot \text{abs}(\text{incx}))\).

incx
Specifies the increment for the elements of \( x \).

y
Array, size at least \((1 + (n -1) \cdot \text{abs}(\text{incy}))\).

incy
Specifies the increment for the elements of \( y \).

Output Parameters

dotc
Contains the result of the dot product of the conjugated \( x \) and unconjugated \( y \), if \( n \) is positive. Otherwise, it contains 0.

cblas_?dotu

Computes a vector-vector dot product.

Syntax

void cblas_cdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);
void cblas_zdotu_sub (const MKL_INT n, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *dotu);

Include Files
• mkl.h

Description
The ?dotu routines perform a vector-vector reduction operation defined as

\[ \text{res} = \sum_{i=1}^{n} x_i \cdot y_i \]

where \( x_i \) and \( y_i \) are elements of complex vectors \( x \) and \( y \).

Input Parameters

\( n \) Specifies the number of elements in vectors \( x \) and \( y \).
\( x \) Array, size at least \( (1 + (n-1) \cdot \text{abs(incx)}) \).
\( \text{incx} \) Specifies the increment for the elements of \( x \).
\( y \) Array, size at least \( (1 + (n-1) \cdot \text{abs(incy)}) \).
\( \text{incy} \) Specifies the increment for the elements of \( y \).

Output Parameters

\( \text{dotu} \) Contains the result of the dot product of \( x \) and \( y \), if \( n \) is positive. Otherwise, it contains 0.

cblas_?nrm2

Computes the Euclidean norm of a vector.

Syntax

float cblas_snrm2 (const MKL_INT n, const float *x, const MKL_INT incx);
double cblas_dnrm2 (const MKL_INT n, const double *x, const MKL_INT incx);
float cblas_scnrm2 (const MKL_INT n, const void *x, const MKL_INT incx);
double cblas_dznrm2 (const MKL_INT n, const void *x, const MKL_INT incx);

Include Files
• mkl.h

Description
The ?nrm2 routines perform a vector reduction operation defined as

\[ \text{res} = ||x||, \]

where:
\( x \) is a vector;
\( \text{res} \) is a value containing the Euclidean norm of the elements of \( x \).
Input Parameters

\( n \)  
Specifies the number of elements in vector \( x \).

\( x \)  
Array, size at least \((1 + (n - 1) \cdot \text{abs (incx)})\).

\( \text{incx} \)  
Specifies the increment for the elements of \( x \).

Return Values

The Euclidean norm of the vector \( x \).

\texttt{cblas\_?rot}

\textit{Performs rotation of points in the plane.}

Syntax

\begin{verbatim}
void cblas_srot (const MKL_INT n, float *x, const MKL_INT incx, float *y, const MKL_INT incy, const float c, const float s);
void cblas_drot (const MKL_INT n, double *x, const MKL_INT incx, double *y, const MKL_INT incy, const double c, const double s);
void cblas csrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy, const float c, const float s);
void cblas zdrot (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy, const double c, const double s);
\end{verbatim}

Include Files

- mkl.h

Description

Given two complex vectors \( x \) and \( y \), each vector element of these vectors is replaced as follows:

\[
\begin{align*}
  x_i &= c \cdot x_i + s \cdot y_i \\
  y_i &= c \cdot y_i - s \cdot x_i
\end{align*}
\]

Input Parameters

\( n \)  
Specifies the number of elements in vectors \( x \) and \( y \).

\( x \)  
Array, size at least \((1 + (n-1) \cdot \text{abs (incx)})\).

\( \text{incx} \)  
Specifies the increment for the elements of \( x \).

\( y \)  
Array, size at least \((1 + (n-1) \cdot \text{abs (incy)})\).

\( \text{incy} \)  
Specifies the increment for the elements of \( y \).

\( c \)  
A scalar.

\( s \)  
A scalar.

Output Parameters

\( x \)  
Each element is replaced by \( c \cdot x + s \cdot y \).
Each element is replaced by \( c^*y - s^*x \).

**cblas_?rotg**  
*Computes the parameters for a Givens rotation.*

**Syntax**
void cblas_srotg (float *a, float *b, float *c, float *s);
void cblas_drotg (double *a, double *b, double *c, double *s);
void cblas_crotg (void *a, const void *b, float *c, void *s);
void cblas_zrotg (void *a, const void *b, double *c, void *s);

**Include Files**
- mkl.h

**Description**
Given the Cartesian coordinates \((a, b)\) of a point, these routines return the parameters \( c, s, r, \) and \( z \) associated with the Givens rotation. The parameters \( c \) and \( s \) define a unitary matrix such that:

\[
\begin{bmatrix}
  c & s \\
 r & -c \\
\end{bmatrix}
\begin{bmatrix}
  a \\
 b \\
\end{bmatrix}
= \begin{bmatrix}
  r \\
  0 \\
\end{bmatrix}
\]

The parameter \( z \) is defined such that if \(|a| > |b|\), \( z \) is \( s \); otherwise if \( c \) is not 0 \( z \) is \( 1/c \); otherwise \( z \) is 1.

**Input Parameters**
- \( a \)  
  Provides the \( x \)-coordinate of the point \( p \).
- \( b \)  
  Provides the \( y \)-coordinate of the point \( p \).

**Output Parameters**
- \( a \)  
  Contains the parameter \( r \) associated with the Givens rotation.
- \( b \)  
  Contains the parameter \( z \) associated with the Givens rotation.
- \( c \)  
  Contains the parameter \( c \) associated with the Givens rotation.
- \( s \)  
  Contains the parameter \( s \) associated with the Givens rotation.

**cblas_?rotm**  
*Performs modified Givens rotation of points in the plane.*

**Syntax**
void cblas_srotm (const MKL_INT n, float *x, const MKL_INT incx, float *y, const MKL_INT incy, const float *param);
void cblas_drotm (const MKL_INT n, double *x, const MKL_INT incx, double *y, const MKL_INT incy, const double *param);

**Include Files**
- mkl.h
Description

Given two vectors $x$ and $y$, each vector element of these vectors is replaced as follows:

$$
\begin{bmatrix}
  x_i \\
  y_i 
\end{bmatrix} = H
\begin{bmatrix}
  x_i \\
  y_i 
\end{bmatrix}
$$

for $i=1$ to $n$, where $H$ is a modified Givens transformation matrix whose values are stored in the $param[1]$ through $param[4]$ array. See discussion on the $param$ argument.

Input Parameters

$n$  
Specifies the number of elements in vectors $x$ and $y$.

$x$  
Array, size at least $(1 + (n-1) \times \text{abs}(\text{incx}))$.

$incx$  
Specifies the increment for the elements of $x$.

$y$  
Array, size at least $(1 + (n-1) \times \text{abs}(\text{incy}))$.

$incy$  
Specifies the increment for the elements of $y$.

$param$  
The elements of the $param$ array are:

- $param[0]$ contains a switch, $flag$. $param[1-4]$ contain $h_{11}$, $h_{21}$, $h_{12}$, and $h_{22}$, respectively, the components of the array $H$.

Depending on the values of $flag$, the components of $H$ are set as follows:

- $flag = -1.0$: $H = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix}$
- $flag = 0.0$: $H = \begin{bmatrix} 1.0 & h_{12} \\ h_{21} & 1.0 \end{bmatrix}$
- $flag = 1.0$: $H = \begin{bmatrix} h_{11} & 1.0 \\ -1.0 & h_{22} \end{bmatrix}$
- $flag = -2.0$: $H = \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix}$

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of $flag$ and are not required to be set in the $param$ vector.

Output Parameters

$x$  
Each element $x[i]$ is replaced by $h_{11} \times x[i] + h_{12} \times y[i]$.

$y$  
Each element $y[i]$ is replaced by $h_{21} \times x[i] + h_{22} \times y[i]$.

$cblas\_rotmg$

Computes the parameters for a modified Givens rotation.
Syntax

```c
void cblas_srotmg (float *d1, float *d2, float *x1, const float y1, float *param);
void cblas_drotmg (double *d1, double *d2, double *x1, const double y1, double *param);
```

Include Files

- mkl.h

Description

Given Cartesian coordinates \((x1, y1)\) of an input vector, these routines compute the components of a modified Givens transformation matrix \(H\) that zeros the \(y\)-component of the resulting vector:

\[
\begin{bmatrix}
x1 \\
0 \\
\end{bmatrix}
= H \begin{bmatrix}
x1 \sqrt{d1} \\
y1 \sqrt{d2} \\
\end{bmatrix}
\]

Input Parameters

- \(d1\): Provides the scaling factor for the \(x\)-coordinate of the input vector.
- \(d2\): Provides the scaling factor for the \(y\)-coordinate of the input vector.
- \(x1\): Provides the \(x\)-coordinate of the input vector.
- \(y1\): Provides the \(y\)-coordinate of the input vector.

Output Parameters

- \(d1\): Provides the first diagonal element of the updated matrix.
- \(d2\): Provides the second diagonal element of the updated matrix.
- \(x1\): Provides the \(x\)-coordinate of the rotated vector before scaling.
- \(param\): Array, size 5.

The elements of the \(param\) array are:

- \(param[0]\) contains a switch, \(flag\). the other array elements \(param[1-4]\) contain the components of the array \(H\): \(h_{11}, h_{21}, h_{12}, \) and \(h_{22}, \) respectively.

Depending on the values of \(flag\), the components of \(H\) are set as follows:

- \(flag = -1.0: H = \begin{bmatrix}
h_{11} & h_{12} \\
h_{21} & h_{22} \\
\end{bmatrix}\)
- \(flag = 0.0: H = \begin{bmatrix}
1.0 & h_{12} \\
h_{21} & 1.0 \\
\end{bmatrix}\)
- \(flag = 1.0: H = \begin{bmatrix}
h_{11} & 1.0 \\
-1.0 & h_{22} \\
\end{bmatrix}\)
- \(flag = -2.0: H = \begin{bmatrix}
1.0 & 0.0 \\
0.0 & 1.0 \\
\end{bmatrix}\)

In the last three cases, the matrix entries of 1.0, -1.0, and 0.0 are assumed based on the value of \(flag\) and are not required to be set in the \(param\) vector.
cblas_?scal

Computes the product of a vector by a scalar.

Syntax

```c
void cblas_sscal (const MKL_INT n, const float a, float *x, const MKL_INT incx);
void cblas_dscal (const MKL_INT n, const double a, double *x, const MKL_INT incx);
void cblas_cscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_zscal (const MKL_INT n, const void *a, void *x, const MKL_INT incx);
void cblas_csscal (const MKL_INT n, const float a, void *x, const MKL_INT incx);
void cblas_zdscal (const MKL_INT n, const double a, void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

The ?scal routines perform a vector operation defined as

\[ x = a \times x \]

where:

- \( a \) is a scalar, \( x \) is an \( n \)-element vector.

Input Parameters

- \( n \)
  - Specifies the number of elements in vector \( x \).
- \( a \)
  - Specifies the scalar \( a \).
- \( x \)
  - Array, size at least \( 1 + (n - 1) \times \text{abs}(\text{incx}) \).
- \( \text{incx} \)
  - Specifies the increment for the elements of \( x \).

Output Parameters

- \( x \)
  - Updated vector \( x \).

cblas_?swap

Swaps a vector with another vector.

Syntax

```c
void cblas_sswap (const MKL_INT n, float *x, const MKL_INT incx, float *y, const MKL_INT incy);
void cblas_dswap (const MKL_INT n, double *x, const MKL_INT incx, double *y, const MKL_INT incy);
void cblas_cswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy);
void cblas_zswap (const MKL_INT n, void *x, const MKL_INT incx, void *y, const MKL_INT incy);
```
Include Files

- mkl.h

Description

Given two vectors \( x \) and \( y \), the \( \text{swap} \) routines return vectors \( y \) and \( x \) swapped, each replacing the other.

**Input Parameters**

- \( n \): Specifies the number of elements in vectors \( x \) and \( y \).
- \( x \): Array, size at least \( (1 + (n-1) \times \text{abs}(\text{incx})) \).
- \( \text{incx} \): Specifies the increment for the elements of \( x \).
- \( y \): Array, size at least \( (1 + (n-1) \times \text{abs}(\text{incy})) \).
- \( \text{incy} \): Specifies the increment for the elements of \( y \).

**Output Parameters**

- \( x \): Contains the resultant vector \( x \), that is, the input vector \( y \).
- \( y \): Contains the resultant vector \( y \), that is, the input vector \( x \).

**cblas_i?amax**

Find the index of the element with maximum absolute value.

**Syntax**

```c
CBLAS_INDEX cblas_isamax (const MKL_INT n, const float *x, const MKL_INT incx);
CBLAS_INDEX cblas_idamax (const MKL_INT n, const double *x, const MKL_INT incx);
CBLAS_INDEX cblas_icamax (const MKL_INT n, const void *x, const MKL_INT incx);
CBLAS_INDEX cblas_izamax (const MKL_INT n, const void *x, const MKL_INT incx);
```

Include Files

- mkl.h

Description

Given a vector \( x \), the \( i?amax \) functions return the position of the vector element \( x[i] \) that has the largest absolute value for real flavors, or the largest sum \( |\text{Re}(x[i])| + |\text{Im}(x[i])| \) for complex flavors.

If either \( n \) or \( \text{incx} \) are not positive, the routine returns 0.

If more than one vector element is found with the same largest absolute value, the index of the first one encountered is returned.

If the vector contains NaN values, then the routine returns the index of the first NaN.

**Input Parameters**

- \( n \): Specifies the number of elements in vector \( x \).
- \( x \): Array, size at least \( (1+(n-1) \times \text{abs}(\text{incx})) \).
incx

Specifies the increment for the elements of \( x \).

Return Values

Returns the position of vector element that has the largest absolute value such that \( x[\text{index}-1] \) has the largest absolute value. The index returned is zero-based.

\texttt{cblas_i?amin}

\textit{Finds the index of the element with the smallest absolute value.}

Syntax

\begin{verbatim}
CBLAS_INDEX cblas_isamin (const MKL_INT \( n \), const float *\( x \), const MKL_INT incx);
CBLAS_INDEX cblas_idamin (const MKL_INT \( n \), const double *\( x \), const MKL_INT incx);
CBLAS_INDEX cblas_icamin (const MKL_INT \( n \), const void *\( x \), const MKL_INT incx);
CBLAS_INDEX cblas_izamin (const MKL_INT \( n \), const void *\( x \), const MKL_INT incx);
\end{verbatim}

Include Files

• mkl.h

Description

Given a vector \( x \), the \texttt{i?amin} functions return the position of the vector element \( x[i] \) that has the smallest absolute value for real flavors, or the smallest sum \(|\text{Re}(x[i])|+|\text{Im}(x[i])|\) for complex flavors.

If either \( n \) or \( incx \) are not positive, the routine returns 0.

If more than one vector element is found with the same smallest absolute value, the index of the first one encountered is returned.

If the vector contains NaN values, then the routine returns the index of the first NaN.

Input Parameters

\begin{itemize}
\item \( n \) On entry, \( n \) specifies the number of elements in vector \( x \).
\item \( x \) Array, size at least \((1+(n-1)*\text{abs}(incx))\).
\item incx Specifies the increment for the elements of \( x \).
\end{itemize}

Return Values

Indicates the position of vector element with the smallest absolute value such that \( x[\text{index}-1] \) has the smallest absolute value. The index returned is zero-based.

\texttt{cblas_?cabs1}

\textit{Computes absolute value of complex number.}

Syntax

\begin{verbatim}
float cblas_scabs1 (const void *\( z \));
double cblas_dcabs1 (const void *\( z \));
\end{verbatim}

Include Files

• mkl.h
Description

The ?cabs1 is an auxiliary routine for a few BLAS Level 1 routines. This routine performs an operation defined as

\[ \text{res} = |\text{Re}(z)| + |\text{Im}(z)|, \]

where \( z \) is a scalar, and \( \text{res} \) is a value containing the absolute value of a complex number \( z \).

Input Parameters

\( z \)  
Scalar.

Return Values

The absolute value of a complex number \( z \).

BLAS Level 2 Routines

This section describes BLAS Level 2 routines, which perform matrix-vector operations. The following table lists the BLAS Level 2 routine groups and the data types associated with them.

<table>
<thead>
<tr>
<th>BLAS Level 2 Routine Groups and Their Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Routine Groups</strong></td>
</tr>
<tr>
<td>cblas_?gbmv</td>
</tr>
<tr>
<td>cblas_?gemv</td>
</tr>
<tr>
<td>cblas_?ger</td>
</tr>
<tr>
<td>cblas_?gerc</td>
</tr>
<tr>
<td>cblas_?geru</td>
</tr>
<tr>
<td>cblas_?hbmv</td>
</tr>
<tr>
<td>cblas_?hemv</td>
</tr>
<tr>
<td>cblas_?her</td>
</tr>
<tr>
<td>cblas_?her2</td>
</tr>
<tr>
<td>cblas_?hpmv</td>
</tr>
<tr>
<td>cblas_?hpr</td>
</tr>
<tr>
<td>cblas_?hpr2</td>
</tr>
<tr>
<td>cblas_?sbmv</td>
</tr>
<tr>
<td>cblas_?spmv</td>
</tr>
<tr>
<td>cblas_?spr</td>
</tr>
<tr>
<td>cblas_?spr2</td>
</tr>
<tr>
<td>cblas_?symv</td>
</tr>
<tr>
<td>cblas_?syr</td>
</tr>
</tbody>
</table>
**Routine Groups**  
**Data Types**  
**Description**  

cblas_?syr2  
s, d  
Rank-2 update of a symmetric matrix  

cblas_?tbmv  
s, d, c, z  
Matrix-vector product using a triangular band matrix  

cblas_?tbsv  
s, d, c, z  
Solution of a linear system of equations with a triangular band matrix  

cblas_?tpmv  
s, d, c, z  
Matrix-vector product using a triangular packed matrix  

cblas_?tpsv  
s, d, c, z  
Solution of a linear system of equations with a triangular packed matrix  

cblas_?trmv  
s, d, c, z  
Matrix-vector product using a triangular matrix  

cblas_?trsv  
s, d, c, z  
Solution of a linear system of equations with a triangular matrix  

cblas_?gbmv  
Computes a matrix-vector product with a general band matrix.  

**Syntax**  
void cblas_sgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);  

void cblas_dgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);  

void cblas_cgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);  

void cblas_zgbmv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT kl, const MKL_INT ku, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);  

**Include Files**  
- mkl.h  

**Description**  
The ?gbmv routines perform a matrix-vector operation defined as  
\[ y := \alpha A x + \beta y, \]  
or  
\[ y := \alpha A^{\prime} x + \beta y, \]  
or  
\[ y := \alpha \text{conjg}(A') x + \beta y, \]  

where:

*alpha* and *beta* are scalars,

*x* and *y* are vectors,

*A* is an *m*-by-*n* band matrix, with *kl* sub-diagonals and *ku* super-diagonals.

**Input Parameters**

*Layout* Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

*trans* Specifies the operation:

- If *trans*=CblasNoTrans, then *y* := *alpha* *A* *x* + *beta* *y*
- If *trans*=CblasTrans, then *y* := *alpha* *A'* *x* + *beta* *y*
- If *trans*=CblasConjTrans, then *y* := *alpha* *conjg(A')* *x* + *beta* *y*

*m* Specifies the number of rows of the matrix *A*.
The value of *m* must be at least zero.

*n* Specifies the number of columns of the matrix *A*.
The value of *n* must be at least zero.

*kl* Specifies the number of sub-diagonals of the matrix *A*.
The value of *kl* must satisfy 0 ≤ *kl*.

*ku* Specifies the number of super-diagonals of the matrix *A*.
The value of *ku* must satisfy 0 ≤ *ku*.

*alpha* Specifies the scalar *alpha*.

*a* Array, size *lda* *n*.

*Layout* = CblasColMajor: Before entry, the leading (*kl* + *ku* + 1) by *n* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied column-by-column, with the leading diagonal of the matrix in row (*ku*) of the array, the first super-diagonal starting at position 1 in row (*ku* − 1), the first sub-diagonal starting at position 0 in row (*ku* + 1), and so on. Elements in the array *a* that do not correspond to elements in the band matrix (such as the top left *ku* by *ku* triangle) are not referenced.

The following program segment transfers a band matrix from conventional full matrix storage (*matrix*, with leading dimension *ldm*) to band storage (*a*, with leading dimension *lda*):

```c
for (j = 0; j < n; j++) {
    k = ku - j;
    for (i = max(0, j-ku); i < min(m, j+kl+1); i++) {
        a[(k+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

*Layout* = CblasRowMajor: Before entry, the leading (*kl* + *ku* + 1) by *m* part of the array *a* must contain the matrix of coefficients. This matrix must be supplied row-by-row, with the leading diagonal of the matrix in column (*kl*) of the array, the first super-diagonal starting at position 0 in column
The first sub-diagonal starting at position 1 in row \((kl - 1)\), and so on. Elements in the array \(a\) that do not correspond to elements in the band matrix (such as the top left \(kl\) by \(kl\) triangle) are not referenced.

The following program segment transfers a band matrix from row-major full matrix storage (\(matrix\), with leading dimension \(ldm\)) to band storage (\(a\), with leading dimension \(lda\)):

```c
for (i = 0; i < m; i++) {
    k = kl - i;
    for (j = max(0, i-kl); j < min(n, i+ku+1); j++) {
        a[(k+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

- \(lda\) Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(lda\) must be at least \((kl + ku + 1)\).

- \(x\) Array, size at least \((1 + (n - 1)*abs(incx))\) when \(trans=CblasNoTrans\), and at least \((1 + (m - 1)*abs(incx))\) otherwise. Before entry, the array \(x\) must contain the vector \(x\).

- \(incx\) Specifies the increment for the elements of \(x\). \(incx\) must not be zero.

- \(beta\) Specifies the scalar \(beta\). When \(beta\) is equal to zero, then \(y\) need not be set on input.

- \(y\) Array, size at least \((1 + (m - 1)*abs(incy))\) when \(trans=CblasNoTrans\) and at least \((1 + (n - 1)*abs(incy))\) otherwise. Before entry, the incremented array \(y\) must contain the vector \(y\).

- \(incy\) Specifies the increment for the elements of \(y\). The value of \(incy\) must not be zero.

### Output Parameters

- \(y\) Buffer holding the updated vector \(y\).

**cblas_?gemv**

*Computes a matrix-vector product using a general matrix.*

### Syntax

```c
void cblas_sgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);
void cblas_dgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);
void cblas_cgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```
void cblas_zgemv (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);

Include Files
• mkl.h

Description
The ?gemv routines perform a matrix-vector operation defined as:

\[ y := \alpha A \star x + \beta y, \]

or

\[ y := \alpha A'^\ast \star x + \beta y, \]

or

\[ y := \alpha \text{conjg}(A'^\ast) \star x + \beta y, \]

where:
alpha and beta are scalars,
x and y are vectors,
A is an m-by-n matrix.

Input Parameters

Layout
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

trans
Specifies the operation:

if trans=CblasNoTrans, then \( y := \alpha A \star x + \beta y \);

if trans=CblasTrans, then \( y := \alpha A'^\ast \star x + \beta y \);

if trans=CblasConjTrans, then \( y := \alpha \text{conjg}(A'^\ast) \star x + \beta y \).

m
Specifies the number of rows of the matrix A. The value of \( m \) must be at least zero.

n
Specifies the number of columns of the matrix A. The value of \( n \) must be at least zero.

alpha
Specifies the scalar \( \alpha \).

a
Array, size lda*\( k \).

For Layout = CblasColMajor, \( k \) is \( n \). Before entry, the leading \( m \)-by-\( n \) part of the array \( a \) must contain the matrix \( A \).

For Layout = CblasRowMajor, \( k \) is \( m \). Before entry, the leading \( n \)-by-\( m \) part of the array \( a \) must contain the matrix \( A \).

lda
Specifies the leading dimension of \( a \) as declared in the calling (sub)program.

For Layout = CblasColMajor, the value of \( lda \) must be at least \( \max(1, m) \).
For \( Layout = \text{CblasRowMajor} \), the value of \( lda \) must be at least \( \max(1,n) \).

\[ x \]

Array, size at least \((1+(n-1)\cdot\text{abs}(\text{incx}))\) when \( \text{trans} = \text{CblasNoTrans} \) and at least \((1+(m-1)\cdot\text{abs}(\text{incx}))\) otherwise. Before entry, the incremented array \( x \) must contain the vector \( x \).

\[ \text{incx} \]

Specifies the increment for the elements of \( x \).

The value of \( \text{incx} \) must not be zero.

\[ \beta \]

Specifies the scalar \( \beta \). When \( \beta \) is set to zero, then \( y \) need not be set on input.

\[ y \]

Array, size at least \((1+(m-1)\cdot\text{abs}(\text{incy}))\) when \( \text{trans} = \text{CblasNoTrans} \) and at least \((1+(n-1)\cdot\text{abs}(\text{incy}))\) otherwise. Before entry with non-zero \( \beta \), the incremented array \( y \) must contain the vector \( y \).

\[ \text{incy} \]

Specifies the increment for the elements of \( y \).

The value of \( \text{incy} \) must not be zero.

**Output Parameters**

\[ y \]

Updated vector \( y \).

**cblas_?ger**

*Performs a rank-1 update of a general matrix.*

**Syntax**

\[
\text{void cblas_sger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy, float *a, const MKL_INT lda);}\\
\text{void cblas_dger (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy, double *a, const MKL_INT lda);}\\
\]

**Include Files**

- mkl.h

**Description**

The \( ? \text{ger} \) routines perform a matrix-vector operation defined as

\[
A := \alpha x^t y + A,
\]

where:

- \( \alpha \) is a scalar,
- \( x \) is an \( m \)-element vector,
- \( y \) is an \( n \)-element vector,
- \( A \) is an \( m \text{-by-} n \) general matrix.
Input Parameters

Layout
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

\(m\)
Specifies the number of rows of the matrix \(A\).
The value of \(m\) must be at least zero.

\(n\)
Specifies the number of columns of the matrix \(A\).
The value of \(n\) must be at least zero.

\(alpha\)
Specifies the scalar \(alpha\).

\(x\)
Array, size at least \((1 + (m - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \(x\) must contain the \(m\)-element vector \(x\).

\(\text{incx}\)
Specifies the increment for the elements of \(x\).
The value of \(\text{incx}\) must not be zero.

\(y\)
Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incy}))\). Before entry, the incremented array \(y\) must contain the \(n\)-element vector \(y\).

\(\text{incy}\)
Specifies the increment for the elements of \(y\).
The value of \(\text{incy}\) must not be zero.

\(a\)
Array, size \(\text{lda} \times k\).

For \(\text{Layout} = \text{CblasColMajor}\), \(k\) is \(n\). Before entry, the leading \(m\)-by-\(n\) part of the array \(a\) must contain the matrix \(A\).

For \(\text{Layout} = \text{CblasRowMajor}\), \(k\) is \(m\). Before entry, the leading \(n\)-by-\(m\) part of the array \(a\) must contain the matrix \(A\).

\(\text{lda}\)
Specifies the leading dimension of \(a\) as declared in the calling (sub)program.

For \(\text{Layout} = \text{CblasColMajor}\), the value of \(\text{lda}\) must be at least \(\max(1, m)\).

For \(\text{Layout} = \text{CblasRowMajor}\), the value of \(\text{lda}\) must be at least \(\max(1, n)\).

Output Parameters

\(a\)
Overwritten by the updated matrix.

\textbf{cblas\_?gerc}
Performs a rank-1 update (conjugated) of a general matrix.

Syntax

```c
void cblas_cgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
```
void cblas_zgerc (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);

Include Files

• mkl.h

Description

The ?gerc routines perform a matrix-vector operation defined as

\[ A := \alpha x^* \text{conj}(y^\prime) + A, \]

where:

\( \alpha \) is a scalar,
\( x \) is an \( m \)-element vector,
\( y \) is an \( n \)-element vector,
\( A \) is an \( m \times n \) matrix.

Input Parameters

- **Layout**: Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
- **m**: Specifies the number of rows of the matrix \( A \).
The value of \( m \) must be at least zero.
- **n**: Specifies the number of columns of the matrix \( A \).
The value of \( n \) must be at least zero.
- **alpha**: Specifies the scalar \( \alpha \).
- **x**: Array, size at least \((1 + (m - 1)\cdot\text{abs(incx)})\). Before entry, the incremented array \( x \) must contain the \( m \)-element vector \( x \).
- **incx**: Specifies the increment for the elements of \( x \).
The value of \( incx \) must not be zero.
- **y**: Array, size at least \((1 + (n - 1)\cdot\text{abs(incy)})\). Before entry, the incremented array \( y \) must contain the \( n \)-element vector \( y \).
- **incy**: Specifies the increment for the elements of \( y \).
The value of \( incy \) must not be zero.
- **a**: Array, size \( lda \cdot k \).
  For \( \text{Layout} = \text{CblasColMajor} \), \( k \) is \( n \). Before entry, the leading \( m \times n \) part of the array \( a \) must contain the matrix \( A \).
  For \( \text{Layout} = \text{CblasRowMajor} \), \( k \) is \( m \). Before entry, the leading \( n \times m \) part of the array \( a \) must contain the matrix \( A \).
- **lda**: Specifies the leading dimension of \( a \) as declared in the calling (sub)program.
For Layout = CblasColMajor, the value of lda must be at least max(1, m).
For Layout = CblasRowMajor, the value of lda must be at least max(1, n).

**Output Parameters**

*a*

Overwritten by the updated matrix.

**cblas_geru**

Performs a rank-1 update (unconjugated) of a general matrix.

**Syntax**

```c
void cblas_cgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
void cblas_zgeru (const CBLAS_LAYOUT Layout, const MKL_INT m, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
```

**Include Files**

- mkl.h

**Description**

The ?geru routines perform a matrix-vector operation defined as

\[ A := \alpha x y' + A, \]

where:

* \( \alpha \) is a scalar,
* \( x \) is an \( m \)-element vector,
* \( y \) is an \( n \)-element vector,
* \( A \) is an \( m \)-by-\( n \) matrix.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
- **m**
  Specifies the number of rows of the matrix \( A \).
  The value of \( m \) must be at least zero.
- **n**
  Specifies the number of columns of the matrix \( A \).
  The value of \( n \) must be at least zero.
- **alpha**
  Specifies the scalar \( \alpha \).
- **x**
  Array, size at least \((1 + (m - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \( x \) must contain the \( m \)-element vector \( x \).
Specifies the increment for the elements of x.
The value of incx must not be zero.

Array, size at least \((1 + (n - 1) \times \text{abs}(incy))\). Before entry, the incremented array y must contain the n-element vector y.

Specifies the increment for the elements of y.
The value of incy must not be zero.

Array, size lda*\(k\).

For Layout = CblasColMajor, \(k\) is \(n\). Before entry, the leading \(m\)-by-\(n\) part of the array a must contain the matrix A.

For Layout = CblasRowMajor, \(k\) is \(m\). Before entry, the leading \(n\)-by-\(m\) part of the array a must contain the matrix A.

Specifies the leading dimension of a as declared in the calling (sub)program.

For Layout = CblasColMajor, the value of lda must be at least \(\max(1, m)\).

For Layout = CblasRowMajor, the value of lda must be at least \(\max(1, n)\).

Output Parameters

Overwritten by the updated matrix.

cblas_?hbmv

Computes a matrix-vector product using a Hermitian band matrix.

Syntax

```c
void cblas_chbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
void cblas_zhbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

Include Files

- mkl.h

Description

The ?hbmv routines perform a matrix-vector operation defined as \(y := alpha*A*x + beta*y\),

where:

- \(alpha\) and \(beta\) are scalars,
- \(x\) and \(y\) are \(n\)-element vectors,
- \(A\) is an \(n\)-by-\(n\) Hermitian band matrix, with \(k\) super-diagonals.
Input Parameters

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**

Specifies whether the upper or lower triangular part of the Hermitian band matrix $A$ is used:

If $\text{uplo} = \text{CblasUpper}$, then the upper triangular part of the matrix $A$ is used.

If $\text{uplo} = \text{CblasLower}$, then the lower triangular part of the matrix $A$ is used.

**n**

Specifies the order of the matrix $A$. The value of $n$ must be at least zero.

**k**

For $\text{uplo} = \text{CblasUpper}$: Specifies the number of super-diagonals of the matrix $A$.

For $\text{uplo} = \text{CblasLower}$: Specifies the number of sub-diagonals of the matrix $A$.

The value of $k$ must satisfy $0 \leq k$.

**alpha**

Specifies the scalar $\alpha$.

**a**

Array, size $\text{lda} \times n$.

$\text{Layout} = \text{CblasColMajor}$:

Before entry with $\text{uplo} = \text{CblasUpper}$, the leading $(k + 1)$ by $n$ part of the array $a$ must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied column-by-column, with the leading diagonal of the matrix in row $k$ of the array, the first super-diagonal starting at position $1$ in row $(k - 1)$, and so on. The top left $k$ by $k$ triangle of the array $a$ is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from conventional full matrix storage ($\text{matrix}$, with leading dimension $\text{ldm}$) to band storage ($a$, with leading dimension $\text{lda}$):

```c
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max( 0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with $\text{uplo} = \text{CblasLower}$, the leading $(k + 1)$ by $n$ part of the array $a$ must contain the lower triangular band part of the Hermitian matrix, supplied column-by-column, with the leading diagonal of the matrix in row $0$ of the array, the first sub-diagonal starting at position $0$ in row $1$, and so on. The bottom right $k$ by $k$ triangle of the array $a$ is not referenced.

The following program segment transfers the lower triangular part of a Hermitian band matrix from conventional full matrix storage ($\text{matrix}$, with leading dimension $\text{ldm}$) to band storage ($a$, with leading dimension $\text{lda}$):

```c
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
```
a[(m+i) + j*lda] = matrix[i + j*ldm];
}
}

Layout = CblasRowMajor:

Before entry with uplo = CblasUpper, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the upper triangular band part of the Hermitian matrix. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage (\(matrix\) with leading dimension \(ldm\)) to row-major band storage (\(a\), with leading dimension \(lda\)):

```c
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

Before entry with uplo = CblasLower, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the lower triangular band part of the Hermitian matrix, supplied row-by-row, with the leading diagonal of the matrix in column \(k\) of the array, the first sub-diagonal starting at position 1 in column \(k-1\), and so on. The top left \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (\(matrix\), with leading dimension \(ldm\)) to row-major band storage (\(a\), with leading dimension \(lda\)):

```c
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

\(lda\)

Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(lda\) must be at least \((k + 1)\).

\(x\)

Array, size at least \((1 + (n - 1)\cdot abs(incx))\). Before entry, the incremented array \(x\) must contain the vector \(x\).

\(incx\)

Specifies the increment for the elements of \(x\).

The value of \(incx\) must not be zero.

\(beta\)

Specifies the scalar \(beta\).

\(y\)

Array, size at least \((1 + (n - 1)\cdot abs(incy))\). Before entry, the incremented array \(y\) must contain the vector \(y\).
Specifies the increment for the elements of $y$.
The value of $incy$ must not be zero.

**Output Parameters**

$y$

Overwritten by the updated vector $y$.

**cblas_?hemv**

*Computes a matrix-vector product using a Hermitian matrix.*

**Syntax**

```c
void cblas_chemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
void cblas_zhemv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *x, const MKL_INT incx, const void *beta, void *y, const MKL_INT incy);
```

**Include Files**

- mkl.h

**Description**

The ?hemv routines perform a matrix-vector operation defined as

$$y := \alpha A x + \beta y,$$

where:

- $\alpha$ and $\beta$ are scalars,
- $x$ and $y$ are $n$-element vectors,
- $A$ is an $n$-by-$n$ Hermitian matrix.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the upper or lower triangular part of the array $a$ is used.
  If $uplo = \text{CblasUpper}$, then the upper triangular of the array $a$ is used.
  If $uplo = \text{CblasLower}$, then the lower triangular of the array $a$ is used.

- **n**
  Specifies the order of the matrix $A$. The value of $n$ must be at least zero.

- **alpha**
  Specifies the scalar $\alpha$.

- **a**
  Array, size $lda\times n$.
  Before entry with $uplo = \text{CblasUpper}$, the leading $n$-by-$n$ upper triangular part of the array $a$ must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of $a$ is not referenced.
entry with uplo = CblasLower, the leading $n$-by-$n$ lower triangular part of the array $a$ must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of $a$ is not referenced.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

$lda$ specifies the leading dimension of $a$ as declared in the calling (sub)program. The value of $lda$ must be at least $\max(1, n)$.

$x$ array, size at least $(1 + (n - 1) \times \text{abs}(\text{incx}))$. Before entry, the incremented array $x$ must contain the $n$-element vector $x$.

$\text{incx}$ specifies the increment for the elements of $x$.

The value of $\text{incx}$ must not be zero.

$\beta$ specifies the scalar $\beta$. When $\beta$ is supplied as zero then $y$ need not be set on input.

$y$ array, size at least $(1 + (n - 1) \times \text{abs}(\text{incy}))$. Before entry, the incremented array $y$ must contain the $n$-element vector $y$.

$\text{incy}$ specifies the increment for the elements of $y$.

The value of $\text{incy}$ must not be zero.

**Output Parameters**

$y$ Overwritten by the updated vector $y$.

**cblas_?her**

*Performs a rank-1 update of a Hermitian matrix.*

**Syntax**

```c
void cblas_cher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
void cblas_zher (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const void *x, const MKL_INT incx, void *a, const MKL_INT lda);
```

**Include Files**

- mkl.h

**Description**

The ?her routines perform a matrix-vector operation defined as

$$ A := \alpha x^* \text{conjg}(x') + A, $$

where:

$\alpha$ is a real scalar,

$x$ is an $n$-element vector,

$A$ is an $n$-by-$n$ Hermitian matrix.
Input Parameters

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**

Specifies whether the upper or lower triangular part of the array $a$ is used.
- If $uplo = \text{CblasUpper}$, then the upper triangular of the array $a$ is used.
- If $uplo = \text{CblasLower}$, then the lower triangular of the array $a$ is used.

**n**

Specifies the order of the matrix $A$. The value of $n$ must be at least zero.

**alpha**

Specifies the scalar $\alpha$.

**x**

Array, dimension at least $(1 + (n - 1) \cdot \text{abs}(incx))$. Before entry, the incremented array $x$ must contain the $n$-element vector $x$.

**incx**

Specifies the increment for the elements of $x$.
- The value of $incx$ must not be zero.

**a**

Array, size $lda \cdot n$.
- Before entry with $uplo = \text{CblasUpper}$, the leading $n$-by-$n$ upper triangular part of the array $a$ must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of $a$ is not referenced.
- Before entry with $uplo = \text{CblasLower}$, the leading $n$-by-$n$ lower triangular part of the array $a$ must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of $a$ is not referenced.
- The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

**lda**

Specifies the leading dimension of $a$ as declared in the calling (sub)program. The value of $lda$ must be at least $\text{max}(1, n)$.

Output Parameters

**a**

- With $uplo = \text{CblasUpper}$, the upper triangular part of the array $a$ is overwritten by the upper triangular part of the updated matrix.
- With $uplo = \text{CblasLower}$, the lower triangular part of the array $a$ is overwritten by the lower triangular part of the updated matrix.
- The imaginary parts of the diagonal elements are set to zero.

**cblas_?her2**

*Performs a rank-2 update of a Hermitian matrix.*

**Syntax**

```c
void cblas_cher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);

void cblas_zher2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *a, const MKL_INT lda);
```
Include Files

- mkl.h

Description

The `her2` routines perform a matrix-vector operation defined as

\[ A := \alpha x^* \text{conjg}(y') + \text{conjg}(\alpha)y^* \text{conjg}(x') + A, \]

where:

- \( \alpha \) is scalar,
- \( x \) and \( y \) are \( n \)-element vectors,
- \( A \) is an \( n \times n \) Hermitian matrix.

Input Parameters

- **Layout**
  - Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  - Specifies whether the upper or lower triangular part of the array \( a \) is used.
  - If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular of the array \( a \) is used.
  - If \( \text{uplo} = \text{CblasLower} \), then the lower triangular of the array \( a \) is used.

- **n**
  - Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **alpha**
  - Specifies the scalar \( \alpha \).

- **x**
  - Array, size at least \( (1 + (n - 1)\times\text{abs}(\text{incx})) \). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).
  - Specifies the increment for the elements of \( x \).
  - The value of \( \text{incx} \) must not be zero.

- **y**
  - Array, size at least \( (1 + (n - 1)\times\text{abs}(\text{incy})) \). Before entry, the incremented array \( y \) must contain the \( n \)-element vector \( y \).
  - Specifies the increment for the elements of \( y \).
  - The value of \( \text{incy} \) must not be zero.

- **a**
  - Array, size \( \text{lda} \times n \).
  - Before entry with \( \text{uplo} = \text{CblasUpper} \), the leading \( n \times n \) upper triangular part of the array \( a \) must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of \( a \) is not referenced.
  - Before entry with \( \text{uplo} = \text{CblasLower} \), the leading \( n \times n \) lower triangular part of the array \( a \) must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of \( a \) is not referenced.
  - The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

- **lda**
  - Specifies the leading dimension of \( a \) as declared in the calling (sub)program. The value of \( \text{lda} \) must be at least \( \max(1, n) \).
Output Parameters

\( a \)

With \( \text{uplo} = \text{CblasUpper} \), the upper triangular part of the array \( a \) is overwritten by the upper triangular part of the updated matrix.

With \( \text{uplo} = \text{CblasLower} \), the lower triangular part of the array \( a \) is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

\texttt{cblas}\_\texttt{?hpmv}

\textit{Computes a matrix-vector product using a Hermitian packed matrix.}

\textbf{Syntax}

\begin{verbatim}
void cblas_chpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, 
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void *beta, 
void *y, const MKL_INT incy);

void cblas_zhpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, 
const void *alpha, const void *ap, const void *x, const MKL_INT incx, const void *beta, 
void *y, const MKL_INT incy);
\end{verbatim}

\textbf{Include Files}

- \texttt{mkl.h}

\textbf{Description}

The \texttt{?hpmv} routines perform a matrix-vector operation defined as

\[ y := \alpha A x + \beta y, \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are \( n \)-element vectors,
- \( A \) is an \( n \)-by-\( n \) Hermitian matrix, supplied in packed form.

\textbf{Input Parameters}

- \textit{Layout}
  - Specifies whether two-dimensional array storage is row-major (\texttt{CblasRowMajor}) or column-major (\texttt{CblasColMajor}).

- \textit{uplo}
  - Specifies whether the upper or lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).

- \textit{n}
  - Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- \textit{alpha}
  - Specifies the scalar \( \alpha \).

- \textit{ap}
  - Array, size at least \( ((n^2(n + 1))/2) \).
For Layout = CblasColMajor:

Before entry with uplo = CblasUpper, the array ap must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that ap[0] contains $A_{1,1}$, ap[1] and ap[2] contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with uplo = CblasLower, the array ap must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that ap[0] contains $A_{1,1}$, ap[1] and ap[2] contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

For Layout = CblasRowMajor:

Before entry with uplo = CblasUpper, the array ap must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, ap[0] contains $A_{1,1}$, ap[1] and ap[2] contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with uplo = CblasLower, the array ap must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that ap[0] contains $A_{1,1}$, ap[1] and ap[2] contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

$x$

Array, size at least $(1 + (n - 1) \times \text{abs}(incx))$. Before entry, the incremented array $x$ must contain the $n$-element vector $x$.

$incx$

Specifies the increment for the elements of $x$.

The value of $incx$ must not be zero.

$beta$

Specifies the scalar $beta$.

When $beta$ is equal to zero then $y$ need not be set on input.

$y$

Array, size at least $(1 + (n - 1) \times \text{abs}(incy))$. Before entry, the incremented array $y$ must contain the $n$-element vector $y$.

$incy$

Specifies the increment for the elements of $y$.

The value of $incy$ must not be zero.

**Output Parameters**

$y$

Overwritten by the updated vector $y$.

**cblas_?hpr**

Performs a rank-1 update of a Hermitian packed matrix.

**Syntax**

```c
void cblas_chpr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const void *x, const MKL_INT incx, void *ap);

void cblas_zhpr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const void *x, const MKL_INT incx, void *ap);
```

**Include Files**

- mkl.h
The `?hpr` routines perform a matrix-vector operation defined as

\[ A := \alpha x^* \text{conjg}(x') + A, \]

where:
- \( \alpha \) is a real scalar,
- \( x \) is an \( n \)-element vector,
- \( A \) is an \( n \times n \) Hermitian matrix, supplied in packed form.

### Input Parameters

#### Layout

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

#### uplo

Specifies whether the upper or lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).

- If \( uplo = \text{CblasUpper} \), the upper triangular part of the matrix \( A \) is supplied in the packed array \( ap \).
- If \( uplo = \text{CblasLower} \), the lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).

#### n

Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

#### alpha

Specifies the scalar \( \alpha \).

#### x

Array, size at least \((1 + (n - 1) \cdot \text{abs}(incx))\). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

#### incx

Specifies the increment for the elements of \( x \). \( incx \) must not be zero.

#### ap

Array, size at least \((n^2 + n)/2\).

For \( \text{Layout} = \text{CblasColMajor} \):

Before entry with \( uplo = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{2,2} \) respectively, and so on.

Before entry with \( uplo = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{3,1} \) respectively, and so on.

For \( \text{Layout} = \text{CblasRowMajor} \):

Before entry with \( uplo = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{1,3} \) respectively, and so on.

Before entry with \( uplo = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{2,2} \) respectively, and so on.
The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

**Output Parameters**

*ap*

With `uplo = CblasUpper`, overwritten by the upper triangular part of the updated matrix.

With `uplo = CblasLower`, overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

**cblas_?hpr2**

*Performs a rank-2 update of a Hermitian packed matrix.*

**Syntax**

```c
void cblas_chpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *ap);
void cblas_zhpr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const void *alpha, const void *x, const MKL_INT incx, const void *y, const MKL_INT incy, void *ap);
```

**Include Files**

- mkl.h

**Description**

The ?hpr2 routines perform a matrix-vector operation defined as

\[ \mathbf{A} := \alpha \mathbf{x} \mathbf{y}^H + \mathbf{y} \mathbf{x}^H \mathbf{A}^H + \mathbf{A}, \]

where:

- \( \alpha \) is a scalar,
- \( \mathbf{x} \) and \( \mathbf{y} \) are \( n \)-element vectors,
- \( \mathbf{A} \) is an \( n \times n \) Hermitian matrix, supplied in packed form.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

- **uplo**
  Specifies whether the upper or lower triangular part of the matrix \( \mathbf{A} \) is supplied in the packed array \( \mathbf{ap} \).
  - If `uplo = CblasUpper`, then the upper triangular part of the matrix \( \mathbf{A} \) is supplied in the packed array \( \mathbf{ap} \).
  - If `uplo = CblasLower`, then the lower triangular part of the matrix \( \mathbf{A} \) is supplied in the packed array \( \mathbf{ap} \).

- **n**
  Specifies the order of the matrix \( \mathbf{A} \). The value of \( n \) must be at least zero.
\textbf{alpha}  
Specifies the scalar \textit{alpha}.

\textbf{x}  
Array, dimension at least \((1 + (n - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \(x\) must contain the \(n\)-element vector \(x\).

\textbf{incx}  
Specifies the increment for the elements of \(x\).  
The value of \(\text{incx}\) must not be zero.

\textbf{y}  
Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incy}))\). Before entry, the incremented array \(y\) must contain the \(n\)-element vector \(y\).

\textbf{incy}  
Specifies the increment for the elements of \(y\).  
The value of \(\text{incy}\) must not be zero.

\textbf{ap}  
Array, size at least \(((n \times (n + 1))/2)\).  

For Layout = \texttt{CblasColMajor}:  
Before entry with \texttt{uplo} = \texttt{CblasUpper}, the array \(ap\) must contain the upper triangular part of the Hermitian matrix packed sequentially, column-by-column, so that \(ap[0]\) contains \(A_{1,1}\), \(ap[1]\) and \(ap[2]\) contain \(A_{1,2}\) and \(A_{2,2}\) respectively, and so on.

Before entry with \texttt{uplo} = \texttt{CblasLower}, the array \(ap\) must contain the lower triangular part of the Hermitian matrix packed sequentially, column-by-column, so that \(ap[0]\) contains \(A_{1,1}\), \(ap[1]\) and \(ap[2]\) contain \(A_{2,1}\) and \(A_{3,1}\) respectively, and so on.

For Layout = \texttt{CblasRowMajor}:  
Before entry with \texttt{uplo} = \texttt{CblasUpper}, the array \(ap\) must contain the upper triangular part of the Hermitian matrix packed sequentially, row-by-row, \(ap[0]\) contains \(A_{1,1}\), \(ap[1]\) and \(ap[2]\) contain \(A_{1,2}\) and \(A_{1,3}\) respectively, and so on.

Before entry with \texttt{uplo} = \texttt{CblasLower}, the array \(ap\) must contain the lower triangular part of the Hermitian matrix packed sequentially, row-by-row, so that \(ap[0]\) contains \(A_{1,1}\), \(ap[1]\) and \(ap[2]\) contain \(A_{2,1}\) and \(A_{2,2}\) respectively, and so on.

The imaginary parts of the diagonal elements need not be set and are assumed to be zero.

\textbf{Output Parameters}  
\textbf{ap}  
With \texttt{uplo} = \texttt{CblasUpper}, overwritten by the upper triangular part of the updated matrix.

With \texttt{uplo} = \texttt{CblasLower}, overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements need are set to zero.

\textbf{cblas_?sbmv}  
 Computes a matrix-vector product with a symmetric band matrix.
**Syntax**

```c
void cblas_ssbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float *x,
const MKL_INT incx, const float beta, float *y, const MKL_INT incy);
void cblas_dsbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n,
const MKL_INT k, const double alpha, const double *a, const MKL_INT lda, const double
*x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);
```

**Include Files**
- mkl.h

**Description**

The `?sbmv` routines perform a matrix-vector operation defined as

\[ y := \alpha \cdot A \cdot x + \beta \cdot y, \]

where:
- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are \( n \)-element vectors,
- \( A \) is an \( n \)-by-\( n \) symmetric band matrix, with \( k \) super-diagonals.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

- **uplo**
  Specifies whether the upper or lower triangular part of the band matrix \( A \) is used:
  - if `uplo = CblasUpper` - upper triangular part;
  - if `uplo = CblasLower` - low triangular part.

- **n**
  Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **k**
  Specifies the number of super-diagonals of the matrix \( A \).
  The value of \( k \) must satisfy \( 0 \leq k \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **a**
  Array, size \( lda \cdot n \). Before entry with `uplo = CblasUpper`, the leading \((k + 1)\) by \( n \) part of the array \( a \) must contain the upper triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row \( k \) of the array, the first super-diagonal starting at position 1 in row \((k - 1)\), and so on. The top left \( k \) by \( k \) triangle of the array \( a \) is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from conventional full matrix storage (`matrix`, with leading dimension `ldm`) to band storage (`a`, with leading dimension `lda`):

```c
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max(0, j - k); i <= j; i++) {
```
Before entry with `uplo = CblasLower`, the leading \((k + 1)\) by \(n\) part of the array \(a\) must contain the lower triangular band part of the symmetric matrix, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right \(k\) by \(k\) triangle of the array \(a\) is not referenced.

The following program segment transfers the lower triangular part of a symmetric band matrix from conventional full matrix storage (`matrix`, with leading dimension `ldm`) to band storage (`a`, with leading dimension `lda`):

```c
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

**Layout = CblasRowMajor:**

Before entry with `uplo = CblasUpper`, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the upper triangular band part of the symmetric matrix. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the upper triangular part of a symmetric band matrix from row-major full matrix storage (`matrix` with leading dimension `ldm`) to row-major band storage (`a`, with leading dimension `lda`):

```c
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

Before entry with `uplo = CblasLower`, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the lower triangular band part of the symmetric matrix, supplied row-by-row, with the leading diagonal of the matrix in column \(k\) of the array, the first sub-diagonal starting at position 1 in column \(k\)-1, and so on. The top left \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the lower triangular part of a symmetric row-major band matrix from row-major full matrix storage (`matrix`, with leading dimension `ldm`) to row-major band storage (`a`, with leading dimension `lda`):

```c
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```
lda Specifies the leading dimension of $a$ as declared in the calling (sub)program. The value of $lda$ must be at least $(k + 1)$.

$x$ Array, size at least $(1 + (n - 1)\times abs(incx))$. Before entry, the incremented array $x$ must contain the vector $x$.

$incx$ Specifies the increment for the elements of $x$.
The value of $incx$ must not be zero.

$beta$ Specifies the scalar $beta$.

$y$ Array, size at least $(1 + (n - 1)\times abs(incy))$. Before entry, the incremented array $y$ must contain the vector $y$.

$incy$ Specifies the increment for the elements of $y$.
The value of $incy$ must not be zero.

Output Parameters

$y$ Overwritten by the updated vector $y$.

**cblas_sspmv**

*Computes a matrix-vector product with a symmetric packed matrix.*

**Syntax**

```c
void cblas_sspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *ap, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);
```

```c
void cblas_dspmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *ap, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);
```

**Include Files**

- mkl.h

**Description**

The ?spmv routines perform a matrix-vector operation defined as

$$ y := alpha \times A \times x + beta \times y, $$

where:

- $alpha$ and $beta$ are scalars,
- $x$ and $y$ are $n$-element vectors,
- $A$ is an $n$-by-$n$ symmetric matrix, supplied in packed form.

**Input Parameters**

*Layout*

- Specifies whether two-dimensional array storage is row-major (**CblasRowMajor**) or column-major (**CblasColMajor**).

*uplo*

- Specifies whether the upper or lower triangular part of the matrix $A$ is supplied in the packed array $ap$. 
If $uplo = \text{CblasUpper}$, then the upper triangular part of the matrix $A$ is supplied in the packed array $ap$.

If $uplo = \text{CblasLower}$, then the lower triangular part of the matrix $A$ is supplied in the packed array $ap$.

$n$ Specifies the order of the matrix $A$. The value of $n$ must be at least zero.

$\alpha$ Specifies the scalar $\alpha$.

$ap$ Array, size at least $((n*(n + 1))/2)$.

For $Layout = \text{CblasColMajor}$:

Before entry with $uplo = \text{CblasUpper}$, the array $ap$ must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{1,2}$ and $A_{2,2}$ respectively, and so on. Before entry with $uplo = \text{CblasLower}$, the array $ap$ must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{2,1}$ and $A_{3,1}$ respectively, and so on.

For $Layout = \text{CblasRowMajor}$:

Before entry with $uplo = \text{CblasUpper}$, the array $ap$ must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{1,2}$ and $A_{1,3}$ respectively, and so on. Before entry with $uplo = \text{CblasLower}$, the array $ap$ must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that $ap[0]$ contains $A_{1,1}$, $ap[1]$ and $ap[2]$ contain $A_{2,1}$ and $A_{2,2}$ respectively, and so on.

$x$ Array, size at least $(1 + (n - 1)\cdot\text{abs}(incx))$. Before entry, the incremented array $x$ must contain the $n$-element vector $x$.

$incx$ Specifies the increment for the elements of $x$.

The value of $incx$ must not be zero.

$\beta$ Specifies the scalar $\beta$.

When $\beta$ is supplied as zero, then $y$ need not be set on input.

$y$ Array, size at least $(1 + (n - 1)\cdot\text{abs}(incy))$. Before entry, the incremented array $y$ must contain the $n$-element vector $y$.

$incy$ Specifies the increment for the elements of $y$.

The value of $incy$ must not be zero.

Output Parameters

$y$ Overwritten by the updated vector $y$.

cblas_?spr Performs a rank-1 update of a symmetric packed matrix.
Syntax
void cblas_sspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *x, const MKL_INT incx, float *ap);
void cblas_dspr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *x, const MKL_INT incx, double *ap);

Include Files
• mkl.h

Description
The ?spr routines perform a matrix-vector operation defined as

\[ a := \alpha x^\top x^\top + A, \]

where:
- \( \alpha \) is a real scalar,
- \( x \) is an \( n \)-element vector,
- \( A \) is an \( n \)-by-\( n \) symmetric matrix, supplied in packed form.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (\texttt{CblasRowMajor}) or column-major (\texttt{CblasColMajor}).
- **uplo**
  Specifies whether the upper or lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).
  - If \( uplo = \texttt{CblasUpper} \), then the upper triangular part of the matrix \( A \) is supplied in the packed array \( ap \).
  - If \( uplo = \texttt{CblasLower} \), then the lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).
- **n**
  Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.
- **alpha**
  Specifies the scalar \( \alpha \).
- **x**
  Array, size at least \( (1 + (n - 1) \times \text{abs(incx)}) \). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).
  Specifies the increment for the elements of \( x \).
- **incx**
  The value of \( incx \) must not be zero.
- **ap**
  For \( \text{Layout} = \texttt{CblasColMajor} \):
  Before entry with \( uplo = \texttt{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{2,2} \) respectively, and so on.
  Before entry with \( uplo = \texttt{CblasLower} \), the array \( ap \) must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{3,1} \) respectively, and so on.
For \( \text{Layout} = \text{CblasRowMajor} \):

Before entry with \( \text{uplo} = \text{CblasUpper} \), the array \( \text{ap} \) must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, \( \text{ap}[0] \) contains \( A_{1,1} \), \( \text{ap}[1] \) and \( \text{ap}[2] \) contain \( A_{1,2} \) and \( A_{1,3} \) respectively, and so on.

Before entry with \( \text{uplo} = \text{CblasLower} \), the array \( \text{ap} \) must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that \( \text{ap}[0] \) contains \( A_{1,1} \), \( \text{ap}[1] \) and \( \text{ap}[2] \) contain \( A_{2,1} \) and \( A_{2,2} \) respectively, and so on.

**Output Parameters**

\( \text{ap} \)

With \( \text{uplo} = \text{CblasUpper} \), overwritten by the upper triangular part of the updated matrix.

With \( \text{uplo} = \text{CblasLower} \), overwritten by the lower triangular part of the updated matrix.

**cblas_?spr2**

*Computes a rank-2 update of a symmetric packed matrix.*

**Syntax**

```c
void cblas_sspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy, float *ap);
void cblas_dspr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy, double *ap);
```

**Include Files**

- mkl.h

**Description**

The \(?spr2\) routines perform a matrix-vector operation defined as

\[
A := \alpha x^* y^* + \alpha y^* x^* + A,
\]

where:

- \( \alpha \) is a scalar,
- \( x \) and \( y \) are \( n \)-element vectors,
- \( A \) is an \( n \)-by-\( n \) symmetric matrix, supplied in packed form.

**Input Parameters**

- **Layout**
  - Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  - Specifies whether the upper or lower triangular part of the matrix \( A \) is supplied in the packed array \( \text{ap} \).
If \( uplo = \text{CblasUpper} \), then the upper triangular part of the matrix \( A \) is supplied in the packed array \( ap \).

If \( uplo = \text{CblasLower} \), then the lower triangular part of the matrix \( A \) is supplied in the packed array \( ap \).

\( n \)

Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

\( \alpha \)

Specifies the scalar \( \alpha \).

\( x \)

Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

\( \text{incx} \)

Specifies the increment for the elements of \( x \).

The value of \( \text{incx} \) must not be zero.

\( y \)

Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incy}))\). Before entry, the incremented array \( y \) must contain the \( n \)-element vector \( y \).

\( \text{incy} \)

Specifies the increment for the elements of \( y \). The value of \( \text{incy} \) must not be zero.

\( \text{ap} \)

For \( \text{Layout} = \text{CblasColMajor} \):

Before entry with \( uplo = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the symmetric matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{2,2} \) respectively, and so on.

Before entry with \( uplo = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the symmetric matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{3,1} \) respectively, and so on.

For \( \text{Layout} = \text{CblasRowMajor} \):

Before entry with \( uplo = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the symmetric matrix packed sequentially, row-by-row, \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{1,3} \) respectively, and so on.

Before entry with \( uplo = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the symmetric matrix packed sequentially, row-by-row, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{2,2} \) respectively, and so on.

Output Parameters

\( \text{ap} \)

With \( uplo = \text{CblasUpper} \), overwritten by the upper triangular part of the updated matrix.

With \( uplo = \text{CblasLower} \), overwritten by the lower triangular part of the updated matrix.

\texttt{cblas\_symv}

\textit{Computes a matrix-vector product for a symmetric matrix.}
Syntax

void cblas_ssymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const float *x, const MKL_INT incx, const float beta, float *y, const MKL_INT incy);

void cblas_dsymv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const double *x, const MKL_INT incx, const double beta, double *y, const MKL_INT incy);

Include Files

• mkl.h

Description

The ?symv routines perform a matrix-vector operation defined as

\[ y := \alpha A x + \beta y, \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are \( n \)-element vectors,
- \( A \) is an \( n \times n \) symmetric matrix.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the upper or lower triangular part of the array \( a \) is used.
  
  If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the array \( a \) is used.
  
  If \( \text{uplo} = \text{CblasLower} \), then the lower triangular part of the array \( a \) is used.

- **n**
  Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **alpha**
  Specifies the scalar \( \alpha \).

- **a**
  Array, size \( lda \times n \).
  
  Before entry with \( \text{uplo} = \text{CblasUpper} \), the leading \( n \times n \) upper triangular part of the array \( a \) must contain the upper triangular part of the symmetric matrix \( A \) and the strictly lower triangular part of \( a \) is not referenced. Before entry with \( \text{uplo} = \text{CblasLower} \), the leading \( n \times n \) lower triangular part of the array \( a \) must contain the lower triangular part of the symmetric matrix \( A \) and the strictly upper triangular part of \( a \) is not referenced.

- **lda**
  Specifies the leading dimension of \( a \) as declared in the calling (sub)program. The value of \( lda \) must be at least \( \max(1, n) \).

- **x**
  Array, size at least \( (1 + (n - 1) \times \text{abs}(\text{incx})) \). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

- **incx**
  Specifies the increment for the elements of \( x \).
  
  The value of \( \text{incx} \) must not be zero.
beta Specifies the scalar beta. When beta is supplied as zero, then y need not be set on input.

y Array, size at least (1 + (n - 1)*abs(incy)). Before entry, the incremented array y must contain the n-element vector y.

incy Specifies the increment for the elements of y. The value of incy must not be zero.

Output Parameters
y Overwritten by the updated vector y.

cblas_?syr
Performs a rank-1 update of a symmetric matrix.

Syntax
void cblas_ssyr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *x, const MKL_INT incx, float *a, const MKL_INT lda);
void cblas_dsyr (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *x, const MKL_INT incx, double *a, const MKL_INT lda);

Include Files
• mkl.h

Description
The ?syr routines perform a matrix-vector operation defined as
A := alpha*x*x' + A,
where:
alpha is a real scalar,
x is an n-element vector,
A is an n-by-n symmetric matrix.

Input Parameters
Layout Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

uplo Specifies whether the upper or lower triangular part of the array a is used.
If uplo = CblasUpper, then the upper triangular part of the array a is used.
If uplo = CblasLower, then the low triangular part of the array a is used.

n Specifies the order of the matrix A. The value of n must be at least zero.

alpha Specifies the scalar alpha.
Array, size at least \((1 + (n-1) \cdot \text{abs}(\text{incx}))\). Before entry, the incremented array \(x\) must contain the \(n\)-element vector \(x\).

**incx**

Specifies the increment for the elements of \(x\).

The value of \(\text{incx}\) must not be zero.

Array, size \(\text{lda} \cdot n\).

Before entry with \(\text{uplo} = \text{CblasUpper}\), the leading \(n\)-by-\(n\) upper triangular part of the array \(a\) must contain the upper triangular part of the symmetric matrix \(A\) and the strictly lower triangular part of \(a\) is not referenced.

Before entry with \(\text{uplo} = \text{CblasLower}\), the leading \(n\)-by-\(n\) lower triangular part of the array \(a\) must contain the lower triangular part of the symmetric matrix \(A\) and the strictly upper triangular part of \(a\) is not referenced.

Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(\text{lda}\) must be at least \(\max(1, n)\).

**Output Parameters**

Array, size \(\text{lda} \cdot n\).

With \(\text{uplo} = \text{CblasUpper}\), the upper triangular part of the array \(a\) is overwritten by the upper triangular part of the updated matrix.

With \(\text{uplo} = \text{CblasLower}\), the lower triangular part of the array \(a\) is overwritten by the lower triangular part of the updated matrix.

**cblas_?syr2**

*Performs a rank-2 update of a symmetric matrix.*

**Syntax**

```c
void cblas_ssyr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const float alpha, const float *x, const MKL_INT incx, const float *y, const MKL_INT incy, float *a, const MKL_INT lda);

void cblas_dsyr2 (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const MKL_INT n, const double alpha, const double *x, const MKL_INT incx, const double *y, const MKL_INT incy, double *a, const MKL_INT lda);
```

**Include Files**

- mkl.h

**Description**

The ?syr2 routines perform a matrix-vector operation defined as

\[ A := \alpha x^* y' + \alpha y^* x' + A, \]

where:

- \(\alpha\) is scalar,
- \(x\) and \(y\) are \(n\)-element vectors,
- \(A\) is an \(n\)-by-\(n\) symmetric matrix.
Input Parameters

**Layout**
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**
Specifies whether the upper or lower triangular part of the array \( a \) is used.
If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the array \( a \) is used.
If \( \text{uplo} = \text{CblasLower} \), then the low triangular part of the array \( a \) is used.

**n**
Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

**alpha**
Specifies the scalar \( \alpha \).

**x**
Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

**incx**
Specifies the increment for the elements of \( x \).
The value of \( \text{incx} \) must not be zero.

**y**
Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incy}))\). Before entry, the incremented array \( y \) must contain the \( n \)-element vector \( y \).

**incy**
Specifies the increment for the elements of \( y \). The value of \( \text{incy} \) must not be zero.

**a**
Array, size \( lda \times n \).
Before entry with \( \text{uplo} = \text{CblasUpper} \), the leading \( n \)-by-\( n \) upper triangular part of the array \( a \) must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of \( a \) is not referenced.
Before entry with \( \text{uplo} = \text{CblasLower} \), the leading \( n \)-by-\( n \) lower triangular part of the array \( a \) must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of \( a \) is not referenced.

**lda**
Specifies the leading dimension of \( a \) as declared in the calling (sub)program. The value of \( \text{lda} \) must be at least \( \max(1, n) \).

Output Parameters

**a**
With \( \text{uplo} = \text{CblasUpper} \), the upper triangular part of the array \( a \) is overwritten by the upper triangular part of the updated matrix.
With \( \text{uplo} = \text{CblasLower} \), the lower triangular part of the array \( a \) is overwritten by the lower triangular part of the updated matrix.

**cblas\_?tbmv**
*Computes a matrix-vector product using a triangular band matrix.*

**Syntax**
```c
void cblas_stbmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const float *a, const MKL_INT lda, float *x, const MKL_INT incx);
```
void cblas_dtbmv (const CBLAS_LAYOUT layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const double *a, const MKL_INT lda, double *x, const MKL_INT incx);
void cblas_ctbmv (const CBLAS_LAYOUT layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);
void cblas_ztbmv (const CBLAS_LAYOUT layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);

Include Files
• mkl.h

Description
The ?tbmv routines perform one of the matrix-vector operations defined as
\[ x := A \cdot x, \text{ or } x := A' \cdot x, \text{ or } x := \text{conjg}(A') \cdot x, \]
where:
- \( x \) is an \( n \)-element vector,
- \( A \) is an \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular band matrix, with \((k + 1)\) diagonals.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the matrix \( A \) is an upper or lower triangular matrix:
  \[
  \text{if } uplo = \text{CblasUpper}, \text{ then the matrix is upper triangular.} \\
  \text{if } uplo = \text{CblasLower}, \text{ then the matrix is lower triangular.} 
  \]

- **trans**
  Specifies the operation:
  \[
  \text{if } \text{trans=} \text{CblasNoTrans}, \text{ then } x := A \cdot x; \\
  \text{if } \text{trans=} \text{CblasTrans}, \text{ then } x := A' \cdot x; \\
  \text{if } \text{trans=} \text{CblasConjTrans}, \text{ then } x := \text{conjg}(A') \cdot x. 
  \]

- **diag**
  Specifies whether the matrix \( A \) is unit triangular:
  \[
  \text{if } diag = \text{CblasUnit} , \text{ then the matrix is unit triangular;} \\
  \text{if } diag = \text{CblasNonUnit} , \text{ then the matrix is not unit triangular.} 
  \]

- **n**
  Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **k**
  On entry with \( uplo = \text{CblasUpper} \), specifies the number of super-diagonals of the matrix \( A \). On entry with \( uplo = \text{CblasLower} \), \( k \) specifies the number of sub-diagonals of the matrix \( a \).
  The value of \( k \) must satisfy \( 0 \leq k \).

- **a**
  Array, size \( lda \cdot n \).
  Layout = CblasColMajor:
Before entry with\texttt{uplo = CblasUpper}, the leading \((k + 1)\) by \(n\) part of the array \(a\) must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row \(k\) of the array, the first super-diagonal starting at position 1 in row \((k - 1)\), and so on. The top left \(k\) by \(k\) triangle of the array \(a\) is not referenced. The following program segment transfers an upper triangular band matrix from conventional full matrix storage (\texttt{matrix}, with leading dimension \texttt{ldm}) to band storage (\texttt{a}, with leading dimension \texttt{lda}): 

```c
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max( 0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with \texttt{uplo = CblasLower}, the leading \((k + 1)\) by \(n\) part of the array \(a\) must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right \(k\) by \(k\) triangle of the array \(a\) is not referenced. The following program segment transfers a lower triangular band matrix from conventional full matrix storage (\texttt{matrix}, with leading dimension \texttt{ldm}) to band storage (\texttt{a}, with leading dimension \texttt{lda}): 

```c
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Note that when \texttt{diag = CblasUnit}, the elements of the array \(a\) corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

Layout = CblasRowMajor:

Before entry with \texttt{uplo = CblasUpper}, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the upper triangular band part of the matrix of coefficients. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage (\texttt{matrix} with leading dimension \texttt{ldm}) to row-major band storage (\texttt{a}, with leading dimension \texttt{lda}): 

```c
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```
Before entry with \texttt{uplo = CblasLower}, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the lower triangular band part of the matrix of coefficients, supplied row-by-row, with the leading diagonal of the matrix in column \(k\) of the array, the first sub-diagonal starting at position 1 in column \(k-1\), and so on. The top left \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (\texttt{matrix}, with leading dimension \(ldm\)) to row-major band storage (\(a\), with leading dimension \(lda\)):

```c
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

\(lda\)

Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(lda\) must be at least \((k + 1)\).

\(x\)

Array, size at least \((1 + (n - 1)\cdot\text{abs}(\text{incx}))\). Before entry, the incremented array \(x\) must contain the \(n\)-element vector \(x\).

\(incx\)

Specifies the increment for the elements of \(x\).

The value of \(incx\) must not be zero.

**Output Parameters**

\(x\)

Overwritten with the transformed vector \(x\).

\texttt{cblas\_?tbsv}

Solves a system of linear equations whose coefficients are in a triangular band matrix.

**Syntax**

```c
void cblas_stbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const float *a, const MKL_INT lda, float *x, const MKL_INT incx);
void cblas_dtbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const double *a, const MKL_INT lda, double *x, const MKL_INT incx);
void cblas_ctbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);
void cblas_ztbsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const MKL_INT k, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

**Include Files**

- mkl.h
Description

The ?tbsv routines solve one of the following systems of equations:

\[ A^{}x = b, \text{ or } A'^{}x = b, \text{ or } \text{conjg}(A')^{}x = b, \]

where:

- \( b \) and \( x \) are \( n \)-element vectors,
- \( A \) is an \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular band matrix, with \((k + 1)\) diagonals.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the matrix \( A \) is an upper or lower triangular matrix:
  - if \( \text{uplo} = \text{CblasUpper} \) the matrix is upper triangular;
  - if \( \text{uplo} = \text{CblasLower} \), the matrix is low triangular.

- **trans**
  Specifies the system of equations:
  - if \( \text{trans} = \text{CblasNoTrans} \), then \( A^{}x = b \);
  - if \( \text{trans} = \text{CblasTrans} \), then \( A'^{}x = b \);
  - if \( \text{trans} = \text{CblasConjTrans} \), then \( \text{conjg}(A')^{}x = b \).

- **diag**
  Specifies whether the matrix \( A \) is unit triangular:
  - if \( \text{diag} = \text{CblasUnit} \) then the matrix is unit triangular;
  - if \( \text{diag} = \text{CblasNonUnit} \), then the matrix is not unit triangular.

- **n**
  Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **k**
  On entry with \( \text{uplo} = \text{CblasUpper} \), \( k \) specifies the number of super-diagonals of the matrix \( A \). On entry with \( \text{uplo} = \text{CblasLower} \), \( k \) specifies the number of sub-diagonals of the matrix \( A \).
  The value of \( k \) must satisfy \( 0 \leq k \).

- **a**
  Array, size \( lda^{}n \).
  
  **Layout = CblasColMajor:**
  Before entry with \( \text{uplo} = \text{CblasUpper} \), the leading \((k + 1)\) by \( n \) part of the array \( a \) must contain the upper triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row \( k \) of the array, the first super-diagonal starting at position 1 in row \((k - 1)\), and so on. The top left \( k \) by \( k \) triangle of the array \( a \) is not referenced.
The following program segment transfers an upper triangular band matrix from conventional full matrix storage (matrix, with leading dimension ldm) to band storage (a, with leading dimension lda):

```c
for (j = 0; j < n; j++) {
    m = k - j;
    for (i = max(0, j - k); i <= j; i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

Before entry with uplo = CblasLower, the leading \((k + 1)\) by \(n\) part of the array \(a\) must contain the lower triangular band part of the matrix of coefficients, supplied column-by-column, with the leading diagonal of the matrix in row 0 of the array, the first sub-diagonal starting at position 0 in row 1, and so on. The bottom right \(k\) by \(k\) triangle of the array \(a\) is not referenced.

The following program segment transfers a lower triangular band matrix from conventional full matrix storage (matrix, with leading dimension ldm) to band storage (a, with leading dimension lda):

```c
for (j = 0; j < n; j++) {
    m = -j;
    for (i = j; i < min(n, j + k + 1); i++) {
        a[(m+i) + j*lda] = matrix[i + j*ldm];
    }
}
```

When diag = CblasUnit, the elements of the array \(a\) corresponding to the diagonal elements of the matrix are not referenced, but are assumed to be unity.

**Layout = CblasRowMajor:**

Before entry with uplo = CblasUpper, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the upper triangular band part of the matrix of coefficients. The matrix must be supplied row-by-row, with the leading diagonal of the matrix in column 0 of the array, the first super-diagonal starting at position 0 in column 1, and so on. The bottom right \(k\)-by-\(k\) triangle of array \(a\) is not referenced.

The following program segment transfers the upper triangular part of a Hermitian band matrix from row-major full matrix storage (matrix with leading dimension ldm) to row-major band storage (a, with leading dimension lda):

```c
for (i = 0; i < n; i++) {
    m = -i;
    for (j = i; j < MIN(n, i+k+1); j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

Before entry with uplo = CblasLower, the leading \((k + 1)\)-by-\(n\) part of array \(a\) must contain the lower triangular band part of the matrix of coefficients, supplied row-by-row, with the leading diagonal of the matrix in column \(k\) of the array, the first sub-diagonal starting at position 1 in column \(k-1\), and so on. The top left \(k\)-by-\(k\) triangle of array \(a\) is not referenced.
The following program segment transfers the lower triangular part of a Hermitian row-major band matrix from row-major full matrix storage (\(matrix\), with leading dimension \(ldm\)) to row-major band storage (\(a\), with leading dimension \(lda\)):

```c
for (i = 0; i < n; i++) {
    m = k - i;
    for (j = max(0, i-k); j <= i; j++) {
        a[(m+j) + i*lda] = matrix[j + i*ldm];
    }
}
```

\(lda\) Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(lda\) must be at least \((k + 1)\).

\(x\) Array, size at least \((1 + (n - 1)*abs(incx))\). Before entry, the incremented array \(x\) must contain the \(n\)-element right-hand side vector \(b\).

\(incx\) Specifies the increment for the elements of \(x\).

The value of \(incx\) must not be zero.

**Output Parameters**

\(x\) Overwritten with the solution vector \(x\).

**cblas_?tpmv**

*Computes a matrix-vector product using a triangular packed matrix.*

**Syntax**

```c
void cblas_stpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float *x, const MKL_INT incx);
void cblas_dtpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double *x, const MKL_INT incx);
void cblas_ctpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x, const MKL_INT incx);
void cblas_ztpmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x, const MKL_INT incx);
```

**Include Files**

- mkl.h

**Description**

The ?tpmv routines perform one of the matrix-vector operations defined as

\[ x := A \cdot x, \text{ or } x := A^\top \cdot x, \text{ or } x := \text{conjg}(A^\top) \cdot x, \]

where:
x is an n-element vector,
A is an n-by-n unit, or non-unit, upper or lower triangular matrix, supplied in packed form.

**Input Parameters**

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**

Specifies whether the matrix A is upper or lower triangular:

- if uplo = CblasUpper, then the matrix is upper triangular.
- if uplo = CblasLower, then the matrix is lower triangular.

**trans**

Specifies the operation:

- if trans=CblasNoTrans, then x := A*x;
- if trans=CblasTrans, then x := A'*x;
- if trans=CblasConjTrans, then x := conjg(A')*x.

**diag**

Specifies whether the matrix A is unit triangular:

- if diag = CblasUnit, then the matrix is unit triangular;
- if diag = CblasNonUnit, then the matrix is not unit triangular.

**n**

Specifies the order of the matrix A. The value of n must be at least zero.

**ap**

Array, size at least ((n*(n + 1))/2).

For Layout = CblasColMajor:

Before entry with uplo = CblasUpper, the array ap must contain the upper triangular matrix packed sequentially, column-by-column, so that respectively, and so on. Before entry with uplo = CblasLower, the array ap must contain the lower triangular matrix packed sequentially, column-by-column, so that respectively, and so on. When diag = CblasUnit, the diagonal elements of a are not referenced, but are assumed to be unity.

For Layout = CblasRowMajor:

Before entry with uplo = CblasUpper, the array ap must contain the upper triangular matrix packed sequentially, row-by-row, so that respectively, and so on. Before entry with uplo = CblasLower, the array ap must contain the lower triangular matrix packed sequentially, row-by-row, so that respectively, and so on.

**x**

Array, size at least (1 + (n - 1)*abs(incx)). Before entry, the incremented array x must contain the n-element vector x.

**incx**

Specifies the increment for the elements of x.

The value of incx must not be zero.
Output Parameters

\( x \)
Overwritten with the transformed vector \( x \).

cblas_?tpsv
Solves a system of linear equations whose coefficients
are in a triangular packed matrix.

Syntax

void cblas_stpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *ap, float
*x, const MKL_INT incx);

void cblas_dtpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *ap, double
*x, const MKL_INT incx);

void cblas_ctpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);

void cblas_ztpsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *ap, void *x,
const MKL_INT incx);

Include Files

- mkl.h

Description

The \( ?tpsv \) routines solve one of the following systems of equations
\[ A^t x = b, \text{ or } A'^* x = b, \text{ or } \text{conjg}(A')^* x = b, \]
where:
\( b \) and \( x \) are \( n \)-element vectors,
\( A \) is an \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular matrix, supplied in packed form.
This routine does not test for singularity or near-singularity.
Such tests must be performed before calling this routine.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the matrix \( A \) is upper or lower triangular:
  \[
  \text{if } uplo = \text{CblasUpper}, \text{ then the matrix is upper triangular.} \\
  \text{if } uplo = \text{CblasLower}, \text{ then the matrix is lower triangular.}
  \]

- **trans**
  Specifies the system of equations:
  \[
  \text{if } trans = \text{CblasNoTrans}, \text{ then } A^t x = b; \\
  \text{if } trans = \text{CblasTrans}, \text{ then } A'^* x = b;
  \]
if \( \text{trans} = \text{CblasConjTrans} \), then \( \text{conjg}(A') \times x = b \).

**diag**

Specifies whether the matrix \( A \) is unit triangular:

- if \( \text{diag} = \text{CblasUnit} \) then the matrix is unit triangular;
- if \( \text{diag} = \text{CblasNonUnit} \), then the matrix is not unit triangular.

**n**

Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

**ap**

Array, size at least \( (\frac{n(n + 1)}{2}) \).

For **Layout** = **CblasColMajor**:

Before entry with \( \text{uplo} = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the triangular matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{2,2} \) respectively, and so on.

Before entry with \( \text{uplo} = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the triangular matrix packed sequentially, column-by-column, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{3,1} \) respectively, and so on.

For **Layout** = **CblasRowMajor**:

Before entry with \( \text{uplo} = \text{CblasUpper} \), the array \( ap \) must contain the upper triangular part of the triangular matrix packed sequentially, row-by-row, \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{1,2} \) and \( A_{1,3} \) respectively, and so on. Before entry with \( \text{uplo} = \text{CblasLower} \), the array \( ap \) must contain the lower triangular part of the triangular matrix packed sequentially, row-by-row, so that \( ap[0] \) contains \( A_{1,1} \), \( ap[1] \) and \( ap[2] \) contain \( A_{2,1} \) and \( A_{2,2} \) respectively, and so on.

When \( \text{diag} = \text{CblasUnit} \), the diagonal elements of \( a \) are not referenced, but are assumed to be unity.

**x**

Array, size at least \( (1 + (n - 1) \times \text{abs}(\text{incx})) \). Before entry, the incremented array \( x \) must contain the \( n \)-element right-hand side vector \( b \).

**incx**

Specifies the increment for the elements of \( x \).

The value of \( \text{incx} \) must not be zero.

**Output Parameters**

**x**

Overwritten with the solution vector \( x \).

**cblas_?trmv**

*Computes a matrix-vector product using a triangular matrix.*

**Syntax**

```c
void cblas_strmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const MKL_INT lda, float *x, const MKL_INT incx);
```
void cblas_dtrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const MKL_INT lda, double *x, const MKL_INT incx);

void cblas_ctrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);

void cblas_ztrmv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);

Include Files

- mkl.h

Description

The trmv routines perform one of the following matrix-vector operations defined as

\[ x := Ax \text{, or } x := A'x \text{, or } x := \text{conjg}(A')x, \]

where:

- \( x \) is an \( n \)-element vector,
- \( A \) is an \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular matrix.

Input Parameters

- **Layout**: Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**: Specifies whether the matrix \( A \) is upper or lower triangular:

  - \( uplo = \text{CblasUpper} \) if \( uplo = \text{CblasLower} \), then the matrix is low triangular.

- **trans**: Specifies the operation:

  - if \( trans = \text{CblasNoTrans} \), then \( x := Ax \);
  - if \( trans = \text{CblasTrans} \), then \( x := A'x \);
  - if \( trans = \text{CblasConjTrans} \), then \( x := \text{conjg}(A')x \).

- **diag**: Specifies whether the matrix \( A \) is unit triangular:

  - if \( diag = \text{CblasUnit} \) then the matrix is unit triangular;
  - if \( diag = \text{CblasNonUnit} \), then the matrix is not unit triangular.

- **n**: Specifies the order of the matrix \( A \). The value of \( n \) must be at least zero.

- **a**: Array, size \( lda \times n \). Before entry with \( uplo = \text{CblasUpper} \), the leading \( n \)-by-\( n \) upper triangular part of the array \( a \) must contain the upper triangular matrix and the strictly lower triangular part of \( a \) is not referenced. Before entry with \( uplo = \text{CblasLower} \), the leading \( n \)-by-\( n \) lower triangular part of the array \( a \) must contain the lower triangular matrix and the strictly upper triangular part of \( a \) is not referenced.

  - When \( diag = \text{CblasUnit} \), the diagonal elements of \( a \) are not referenced either, but are assumed to be unity.
Specifies the leading dimension of \( a \) as declared in the calling (sub)program. The value of \( lda \) must be at least \( \max(1, n) \).

Array, size at least \( (1 + (n - 1) \times \text{abs}(\text{incx})) \). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

Specifies the increment for the elements of \( x \).

The value of \( \text{incx} \) must not be zero.

**Output Parameters**

\( x \)

Overwritten with the transformed vector \( x \).

---

**cblas_?trsv**

Solves a system of linear equations whose coefficients are in a triangular matrix.

**Syntax**

```c
void cblas_strsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const float *a, const MKL_INT lda, float *x, const MKL_INT incx);
void cblas_dtrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const double *a, const MKL_INT lda, double *x, const MKL_INT incx);
void cblas_ctrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);
void cblas_ztrsv (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const CBLAS_DIAG diag, const MKL_INT n, const void *a, const MKL_INT lda, void *x, const MKL_INT incx);
```

**Include Files**

- mkl.h

**Description**

The \( ? \text{trsv} \) routines solve one of the systems of equations:

\[
A \times x = b, \text{ or } A^\prime \times x = b, \text{ or } \text{conjg}(A^\prime) \times x = b,
\]

where:

\( b \) and \( x \) are \( n \)-element vectors,

\( A \) is an \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular matrix.

The routine does not test for singularity or near-singularity.

Such tests must be performed before calling this routine.

**Input Parameters**

- **Layout**

  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
uplo

Specifies whether the matrix $A$ is upper or lower triangular:

- $\text{uplo} = \text{CblasUpper}$
  - if $\text{uplo} = \text{CblasLower}$, then the matrix is lower triangular.

$\text{trans}$

Specifies the systems of equations:

- if $\text{trans} = \text{CblasNoTrans}$, then $A*x = b$;
- if $\text{trans} = \text{CblasTrans}$, then $A'*x = b$;
- if $\text{trans} = \text{CblasConjTrans}$, then $\text{conj}(A')*x = b$.

$\text{diag}$

Specifies whether the matrix $A$ is unit triangular:

- if $\text{diag} = \text{CblasUnit}$ then the matrix is unit triangular;
- if $\text{diag} = \text{CblasNonUnit}$, then the matrix is not unit triangular.

$n$

Specifies the order of the matrix $A$. The value of $n$ must be at least zero.

$a$

Array, size $\text{lda}*n$. Before entry with $\text{uplo} = \text{CblasUpper}$, the leading $n$-by-$n$ upper triangular part of the array $a$ must contain the upper triangular matrix and the strictly lower triangular part of $a$ is not referenced. Before entry with $\text{uplo} = \text{CblasLower}$, the leading $n$-by-$n$ lower triangular part of the array $a$ must contain the lower triangular matrix and the strictly upper triangular part of $a$ is not referenced.

When $\text{diag} = \text{CblasUnit}$, the diagonal elements of $a$ are not referenced either, but are assumed to be unity.

$\text{lda}$

Specifies the leading dimension of $a$ as declared in the calling (sub)program. The value of $\text{lda}$ must be at least $\max(1, n)$.

$x$

Array, size at least $(1 + (n - 1)*\text{abs}(\text{incx}))$. Before entry, the incremented array $x$ must contain the $n$-element right-hand side vector $b$.

$\text{incx}$

Specifies the increment for the elements of $x$.

The value of $\text{incx}$ must not be zero.

Output Parameters

$x$

Overwritten with the solution vector $x$.

BLAS Level 3 Routines

BLAS Level 3 routines perform matrix-matrix operations. The following table lists the BLAS Level 3 routine groups and the data types associated with them.

<table>
<thead>
<tr>
<th>Routine Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{cblas_?gemm}$</td>
<td>$s, d, c, z$</td>
<td>Computes a matrix-matrix product with general matrices.</td>
</tr>
<tr>
<td>$\text{cblas_?hemm}$</td>
<td>$c, z$</td>
<td>Computes a matrix-matrix product where one input matrix is Hermitian.</td>
</tr>
<tr>
<td>$\text{cblas_?herk}$</td>
<td>$c, z$</td>
<td>Performs a Hermitian rank-k update.</td>
</tr>
<tr>
<td>$\text{cblas_?her2k}$</td>
<td>$c, z$</td>
<td>Performs a Hermitian rank-2k update.</td>
</tr>
</tbody>
</table>
## Routine Group

<table>
<thead>
<tr>
<th>Routine Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblas_?symm</td>
<td>s, d, c, z</td>
<td>Computes a matrix-matrix product where one input matrix is symmetric.</td>
</tr>
<tr>
<td>cblas_?syrk</td>
<td>s, d, c, z</td>
<td>Performs a symmetric rank-k update.</td>
</tr>
<tr>
<td>cblas_?syr2k</td>
<td>s, d, c, z</td>
<td>Performs a symmetric rank-2k update.</td>
</tr>
<tr>
<td>cblas_?trmm</td>
<td>s, d, c, z</td>
<td>Computes a matrix-matrix product where one input matrix is triangular.</td>
</tr>
<tr>
<td>cblas_?trsm</td>
<td>s, d, c, z</td>
<td>Solves a triangular matrix equation.</td>
</tr>
</tbody>
</table>

### Symmetric Multiprocessing Version of Intel® MKL

Many applications spend considerable time executing BLAS routines. This time can be scaled by the number of processors available on the system through using the symmetric multiprocessing (SMP) feature built into the Intel® Math Kernel Library. The performance enhancements based on the parallel use of the processors are available without any programming effort on your part.

To enhance performance, the library uses the following methods:

- The BLAS functions are blocked where possible to restructure the code in a way that increases the localization of data reference, enhances cache memory use, and reduces the dependency on the memory bus.
- The code is distributed across the processors to maximize parallelism.

### cblas_?gemm

*Computes a matrix-matrix product with general matrices.*

#### Syntax

```c
void cblas_sgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const MKL_INT ldc);
void cblas_dgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const double alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c, const MKL_INT ldc);
void cblas_cgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
void cblas_zgemm (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

### Include Files

- mkl.h

### Description
The ?gemm routines compute a scalar-matrix-matrix product and add the result to a scalar-matrix product, with general matrices. The operation is defined as

\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C \]

where:

- \text{op}(X) is one of \text{op}(X) = X, or \text{op}(X) = X^T, or \text{op}(X) = X^H,
- \alpha and \beta are scalars,
- \( A, \ B \) and \( C \) are matrices:
  - \text{op}(A) is an \( m \)-by-\( k \) matrix,
  - \text{op}(B) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.

See also:
- ?gemm for the Fortran language interface to this routine
- ?gemm3m, BLAS-like extension routines, that use matrix multiplication for similar matrix-matrix operations

### Input Parameters

**Layout**

Specifies whether two-dimensional array storage is row-major (**CblasRowMajor** or column-major (**CblasColMajor**).

**transa**

Specifies the form of \text{op}(A) used in the matrix multiplication:

- if \( \text{transa} = \text{CblasNoTrans} \), then \text{op}(A) = A;
- if \( \text{transa} = \text{CblasTrans} \), then \text{op}(A) = A^T;
- if \( \text{transa} = \text{CblasConjTrans} \), then \text{op}(A) = A^H.

**transb**

Specifies the form of \text{op}(B) used in the matrix multiplication:

- if \( \text{transb} = \text{CblasNoTrans} \), then \text{op}(B) = B;
- if \( \text{transb} = \text{CblasTrans} \), then \text{op}(B) = B^T;
- if \( \text{transb} = \text{CblasConjTrans} \), then \text{op}(B) = B^H.

**m**

Specifies the number of rows of the matrix \text{op}(A) and of the matrix \( C \). The value of \( m \) must be at least zero.

**n**

Specifies the number of columns of the matrix \text{op}(B) and the number of columns of the matrix \( C \). The value of \( n \) must be at least zero.

**k**

Specifies the number of columns of the matrix \text{op}(A) and the number of rows of the matrix \text{op}(B). The value of \( k \) must be at least zero.

**alpha**

Specifies the scalar \( \alpha \).

<table>
<thead>
<tr>
<th>transa=CblasNoTrans</th>
<th>transa=CblasTrans or transa=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasColMajor</td>
<td>Array, size lda*k. Before entry, the leading ( m )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
<tr>
<td></td>
<td>Array, size lda*m. Before entry, the leading ( k )-by-( m ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
</tbody>
</table>
CblasRowMajor

Array, size lda* m.
Before entry, the leading $k$-by-$m$ part of the array $a$
must contain the matrix $A$.

Array, size lda* k.
Before entry, the leading $m$-by-$k$ part of the array $a$
must contain the matrix $A$.

$lda$

Specifies the leading dimension of $a$ as declared in the calling
(sub)program.

transa=CblasNoTrans  transa=CblasTrans or
transa=CblasConjTrans

$transa$

$transa=CblasNoTrans$
tansa=CblasTrans or
transa=CblasConjTrans

$bdb$

Specifies the leading dimension of $b$ as declared in the calling
(sub)program.

When $transb$=CblasNoTrans, then $ldb$ must be at least $\max(1, k)$,
otherwise $ldb$ must be at least $\max(1, n)$.

beta

Specifies the scalar $beta$. When $beta$ is equal to zero, then $c$
need not be set on input.
c

<table>
<thead>
<tr>
<th>Layout</th>
<th>Array, size ldc by n. Before entry, the leading m-by-n part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout</th>
<th>Array, size ldc by m. Before entry, the leading n-by-m part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasRowMajor</td>
<td>---------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>

ldc

Specifies the leading dimension of c as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout</th>
<th>ldc must be at least max(1, m).</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout</th>
<th>ldc must be at least max(1, n).</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasRowMajor</td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

- **c**
  Overwritten by the m-by-n matrix (alpha*op(A)*op(B) + beta*C).

Example

For examples of routine usage, see these code examples in the Intel® MKL installation directory:

- cblas_sgemm: examples\cblas\source\cblas_sgemmx.c
- cblas_dgemm: examples\cblas\source\cblas_dgemmx.c
- cblas_cgemm: examples\cblas\source\cblas_cgemmx.c
- cblas_zgemm: examples\cblas\source\cblas_zgemmx.c

**cblas_?hemm**

*Computes a matrix-matrix product where one input matrix is Hermitian.*

Syntax

```c
void cblas_chemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
void cblas_zhemm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The ?hemm routines compute a scalar-matrix-matrix product using a Hermitian matrix A and a general matrix B and add the result to a scalar-matrix product using a general matrix C. The operation is defined as

\[ C := \alpha A^* B + \beta C \]
or
\[ C := \alpha A + \beta C \]

where:
\( \alpha \) and \( \beta \) are scalars,
\( A \) is a Hermitian matrix,
\( B \) and \( C \) are \( m \text{-by-} n \) matrices.

**Input Parameters**

**Layout**
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**side**
Specifies whether the Hermitian matrix \( A \) appears on the left or right in the operation as follows:
- if \( \text{side} = \text{CblasLeft} \), then \( C := \alpha A B + \beta C \);
- if \( \text{side} = \text{CblasRight} \), then \( C := \alpha B A + \beta C \).

**uplo**
Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is used:
- If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the Hermitian matrix \( A \) is used.
- If \( \text{uplo} = \text{CblasLower} \), then the lower triangular part of the Hermitian matrix \( A \) is used.

**m**
Specifies the number of rows of the matrix \( C \).
The value of \( m \) must be at least zero.

**n**
Specifies the number of columns of the matrix \( C \).
The value of \( n \) must be at least zero.

**alpha**
Specifies the scalar \( \alpha \).

**a**
Array, size \( lda \times ka \), where \( ka \) is \( m \) when \( \text{side} = \text{CblasLeft} \) and is \( n \) otherwise. Before entry with \( \text{side} = \text{CblasLeft} \), the \( m \text{-by-} m \) part of the array \( a \) must contain the Hermitian matrix, such that when \( \text{uplo} = \text{CblasUpper} \), the leading \( m \text{-by-} m \) upper triangular part of the array \( a \) must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of \( a \) is not referenced, and when \( \text{uplo} = \text{CblasLower} \), the leading \( m \text{-by-} m \) lower triangular part of the array \( a \) must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of \( a \) is not referenced.

Before entry with \( \text{side} = \text{CblasRight} \), the \( n \text{-by-} n \) part of the array \( a \) must contain the Hermitian matrix, such that when \( \text{uplo} = \text{CblasUpper} \), the leading \( n \text{-by-} n \) upper triangular part of the array \( a \) must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of \( a \) is not referenced, and when \( \text{uplo} = \text{CblasLower} \), the leading \( n \text{-by-} n \) lower triangular part of the array \( a \) must contain the lower triangular part of the Hermitian matrix, and the strictly upper triangular part of \( a \) is not referenced. The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.
lda

Specifies the leading dimension of a as declared in the calling (sub) program. When side = CblasLeft then lda must be at least max(1, m), otherwise lda must be at least max(1, n).

b

For Layout = CblasColMajor: array, size ldb*n. The leading m-by-n part of the array b must contain the matrix B.

For Layout = CblasRowMajor: array, size ldb*m. The leading n-by-m part of the array b must contain the matrix B.

ldb

Specifies the leading dimension of b as declared in the calling (sub)program. When Layout = CblasColMajor, ldb must be at least max(1, m); otherwise, ldb must be at least max(1, n).

beta

Specifies the scalar beta.

When beta is supplied as zero, then c need not be set on input.

c

For Layout = CblasColMajor: array, size ldc*n. Before entry, the leading m-by-n part of the array c must contain the matrix C, except when beta is zero, in which case c need not be set on entry.

For Layout = CblasRowMajor: array, size ldc*m. Before entry, the leading n-by-m part of the array c must contain the matrix C, except when beta is zero, in which case c need not be set on entry.

ldc

Specifies the leading dimension of c as declared in the calling (sub)program. When Layout = CblasColMajor, ldc must be at least max(1, m); otherwise, ldc must be at least max(1, n).

Output Parameters

c

Overwritten by the m-by-n updated matrix.

cblas_?herk

Performs a Hermitian rank-k update.

Syntax

void cblas_cherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const float beta, void *c, const MKL_INT ldc);

void cblas_zherk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const void *a, const MKL_INT lda, const double beta, void *c, const MKL_INT ldc);

Include Files

• mkl.h

Description

The ?herk routines perform a rank-k matrix-matrix operation using a general matrix A and a Hermitian matrix C. The operation is defined as:

\[ C := \alpha A \bar{A}^T + \beta C, \]
\[ C := \alpha A^H A + \beta C, \]

where:
- \( \alpha \) and \( \beta \) are real scalars,
- \( C \) is an \( n \times n \) Hermitian matrix,
- \( A \) is an \( n \times k \) matrix in the first case and a \( k \times n \) matrix in the second case.

### Input Parameters

**Layout**

Specifies whether two-dimensional array storage is row-major (\texttt{CblasRowMajor}) or column-major (\texttt{CblasColMajor}).

**uplo**

Specifies whether the upper or lower triangular part of the array \( c \) is used.
- If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the array \( c \) is used.
- If \( \text{uplo} = \text{CblasLower} \), then the lower triangular part of the array \( c \) is used.

**trans**

Specifies the operation:
- If \( \text{trans} = \text{CblasNoTrans} \), then \( C := \alpha A^H A + \beta C \);
- If \( \text{trans} = \text{CblasConjTrans} \), then \( C := \alpha A^H A + \beta C \).

**n**

Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

**k**

With \( \text{trans} = \text{CblasNoTrans} \), \( k \) specifies the number of columns of the matrix \( A \), and with \( \text{trans} = \text{CblasConjTrans} \), \( k \) specifies the number of rows of the matrix \( A \).

The value of \( k \) must be at least zero.

**alpha**

Specifies the scalar \( \alpha \).

**a**

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array, size lda*k.</td>
<td>Before entry, the leading n-by-k part of the array a must contain the matrix A.</td>
<td>Array, size lda*n. Before entry, the leading k-by-n part of the array a must contain the matrix A.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array, size lda*n.</td>
<td>Before entry, the leading k-by-n part of the array a must contain the matrix A.</td>
<td>Array, size lda*k. Before entry, the leading n-by-k part of the array a must contain the matrix A.</td>
</tr>
</tbody>
</table>

**lda**

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda must be at least max(1, n).</td>
<td></td>
<td>lda must be at least max(1, k)</td>
</tr>
</tbody>
</table>
### BLAS and Sparse BLAS Routines

<table>
<thead>
<tr>
<th>Layout</th>
<th>( lda ) must be at least</th>
<th>( lda ) must be at least</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasRowMajor</td>
<td>( \max(1, k) )</td>
<td>( \max(1, n) ).</td>
</tr>
</tbody>
</table>

\( beta \)

Specifies the scalar \( beta \).

\( c \)

Array, size \( ldc \) by \( n \).

Before entry with \( uplo = \) CblasUpper, the leading \( n \)-by-\( n \) upper triangular part of the array \( c \) must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of \( c \) is not referenced.

Before entry with \( uplo = \) CblasLower, the leading \( n \)-by-\( n \) lower triangular part of the array \( c \) must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of \( c \) is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

\( ldc \)

Specifies the leading dimension of \( c \) as declared in the calling (sub)program. The value of \( ldc \) must be at least \( \max(1, n) \).

**Output Parameters**

\( c \)

With \( uplo = \) CblasUpper, the upper triangular part of the array \( c \) is overwritten by the upper triangular part of the updated matrix.

With \( uplo = \) CblasLower, the lower triangular part of the array \( c \) is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

**cblas_\(_\!\!\_\!\!\_\!\!\_\)her2k**

Performs a Hermitian rank-2\( k \) update.

**Syntax**

```c
void cblas_cher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const float beta, void *c, const MKL_INT ldc);
void cblas_zher2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const double beta, void *c, const MKL_INT ldc);
```

**Include Files**

- mkl.h

**Description**

The \(_\!\!\_\!\!\_\!\!\_\!\!\_\)her2k routines perform a rank-2\( k \) matrix-matrix operation using general matrices \( A \) and \( B \) and a Hermitian matrix \( C \). The operation is defined as

\[
C := alpha A B^H + \text{conjg}(alpha) B A^H + beta C
\]
or
\[
C := \alpha A^H B + \text{conjg}(\alpha) B^H A + \beta C
\]

where:
* \(\alpha\) is a scalar and \(\beta\) is a real scalar.
* \(C\) is an \(n\)-by-\(n\) Hermitian matrix.
* \(A\) and \(B\) are \(n\)-by-\(k\) matrices in the first case and \(k\)-by-\(n\) matrices in the second case.

**Input Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout</strong></td>
<td>Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).</td>
</tr>
<tr>
<td><strong>uplo</strong></td>
<td>Specifies whether the upper or lower triangular part of the array (c) is used. If (\text{uplo} = \text{CblasUpper}), then the upper triangular of the array (c) is used. If (\text{uplo} = \text{CblasLower}), then the lower triangular of the array (c) is used.</td>
</tr>
</tbody>
</table>
| **trans** | Specifies the operation:  
| If \(\text{trans} = \text{CblasNoTrans}\), then \(C := \alpha A^H B + \alpha B^H A + \beta C\);  
| If \(\text{trans} = \text{CblasConjTrans}\), then \(C := \alpha A^H B + \alpha B^H A + \beta C\). |
| **n** | Specifies the order of the matrix \(C\). The value of \(n\) must be at least zero. |
| **k** | With \(\text{trans} = \text{CblasNoTrans}\) specifies the number of columns of the matrix \(A\), and with \(\text{trans} = \text{CblasConjTrans}\), \(k\) specifies the number of rows of the matrix \(A\). The value of \(k\) must be at least equal to zero. |
| **alpha** | Specifies the scalar \(\alpha\). |

| **a** | Specifies the leading dimension of \(a\) as declared in the calling (sub)program. |

<table>
<thead>
<tr>
<th>Layout</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td>Array, size (lda) (k). Before entry, the leading (n)-by-(k) part of the array (a) must contain the matrix (A).</td>
<td>Array, size (lda) (n). Before entry, the leading (k)-by-(n) part of the array (a) must contain the matrix (A).</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td>Array, size (lda) (n). Before entry, the leading (k)-by-(n) part of the array (a) must contain the matrix (A).</td>
<td>Array, size (lda) (k). Before entry, the leading (n)-by-(k) part of the array (a) must contain the matrix (A).</td>
</tr>
<tr>
<td>Layout</td>
<td>$lda$ must be at least</td>
<td>$lda$ must be at least</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>CblasColMajor</td>
<td>$\max(1, n)$</td>
<td>$\max(1, k)$</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td>$\max(1, k)$</td>
<td>$\max(1, n)$</td>
</tr>
</tbody>
</table>

$beta$ specifies the scalar $beta$.

<table>
<thead>
<tr>
<th>Layout</th>
<th>$trans$</th>
<th>$trans$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td>Array, size $ldb*k$.</td>
<td>Array, size $ldb*n$.</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td>Array, size $lda*n$.</td>
<td>Array, size $lda*k$.</td>
</tr>
</tbody>
</table>

$ldb$ specifies the leading dimension of $a$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout</th>
<th>$ldb$ must be at least</th>
<th>$ldb$ must be at least</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td>$\max(1, n)$</td>
<td>$\max(1, k)$</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td>$\max(1, k)$</td>
<td>$\max(1, n)$</td>
</tr>
</tbody>
</table>

$c$ is an array, size $ldc$ by $n$. Before entry with $uplo = \text{CblasUpper}$, the leading $n$-by-$n$ upper triangular part of the array $c$ must contain the upper triangular part of the Hermitian matrix and the strictly lower triangular part of $c$ is not referenced.

Before entry with $uplo = \text{CblasLower}$, the leading $n$-by-$n$ lower triangular part of the array $c$ must contain the lower triangular part of the Hermitian matrix and the strictly upper triangular part of $c$ is not referenced.

The imaginary parts of the diagonal elements need not be set, they are assumed to be zero.

$ldc$ specifies the leading dimension of $c$ as declared in the calling (sub)program. The value of $ldc$ must be at least $\max(1, n)$.
Output Parameters

`c`

With `uplo = CblasUpper`, the upper triangular part of the array `c` is overwritten by the upper triangular part of the updated matrix.

With `uplo = CblasLower`, the lower triangular part of the array `c` is overwritten by the lower triangular part of the updated matrix.

The imaginary parts of the diagonal elements are set to zero.

### cblas_?symm

Computes a matrix-matrix product where one input matrix is symmetric.

#### Syntax

```c
void cblas_ssymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const MKL_INT ldc);
void cblas_dsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c, const MKL_INT ldc);
void cblas_csymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
void cblas_zsymm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

#### Include Files

- mkl.h

#### Description

The `?symm` routines compute a scalar-matrix-matrix product with one symmetric matrix and add the result to a scalar-matrix product. The operation is defined as

\[ C := \alpha A B + \beta C, \]

or

\[ C := \alpha B A + \beta C, \]

where:

- `alpha` and `beta` are scalars,
- `A` is a symmetric matrix,
- `B` and `C` are `m`-by-`n` matrices.

#### Input Parameters

`Layout`

Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).
Specifies whether the symmetric matrix $A$ appears on the left or right in the operation:

- If $side = \text{CblasLeft}$, then $C := \alpha A^T B + \beta C$;
- If $side = \text{CblasRight}$, then $C := \alpha B^T A + \beta C$.

Specifies whether the upper or lower triangular part of the symmetric matrix $A$ is used:

- If $uplo = \text{CblasUpper}$, then the upper triangular part is used;
- If $uplo = \text{CblasLower}$, then the lower triangular part is used.

Specifies the number of rows of the matrix $C$.

The value of $m$ must be at least zero.

Specifies the number of columns of the matrix $C$.

The value of $n$ must be at least zero.

Specifies the scalar $\alpha$.

Array, size $lda \times ka$, where $ka$ is $m$ when $side = \text{CblasLeft}$ and is $n$ otherwise.

Before entry with $side = \text{CblasLeft}$, the $m$-by-$m$ part of the array $a$ must contain the symmetric matrix, such that when $uplo = \text{CblasUpper}$, the leading $m$-by-$m$ upper triangular part of the array $a$ must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of $a$ is not referenced, and when $side = \text{CblasLeft}$, the leading $m$-by-$m$ lower triangular part of the array $a$ must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of $a$ is not referenced.

Before entry with $side = \text{CblasRight}$, the $n$-by-$n$ part of the array $a$ must contain the symmetric matrix, such that when $uplo = \text{CblasUpper}$, the array $a$ must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of $a$ is not referenced, and when $side = \text{CblasLeft}$, the leading $n$-by-$n$ lower triangular part of the array $a$ must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of $a$ is not referenced.

Specifies the leading dimension of $a$ as declared in the calling (sub)program. When $side = \text{CblasLeft}$ then $lda$ must be at least $\max(1, m)$, otherwise $lda$ must be at least $\max(1, n)$.

For $Layout = \text{CblasColMajor}$: array, size $ldb \times n$. The leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

For $Layout = \text{CblasRowMajor}$: array, size $ldb \times m$. The leading $n$-by-$m$ part of the array $b$ must contain the matrix $B$.

Specifies the leading dimension of $b$ as declared in the calling (sub)program. When $Layout = \text{CblasColMajor}$, $ldb$ must be at least $\max(1, m)$; otherwise, $ldb$ must be at least $\max(1, n)$.

Specifies the scalar $\beta$.

When $\beta$ is set to zero, then $C$ need not be set on input.
cblas_syrk
Performs a symmetric rank-k update.

Syntax

```c
void cblas_ssyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo,
const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float beta, float *c, const MKL_INT ldc);
```

```c
void cblas_dsyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const
double *a, const MKL_INT lda, const double beta, double *c, const MKL_INT ldc);
```

```c
void cblas_csyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
```

```c
void cblas_zsyrk (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const
CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void
*a, const MKL_INT lda, const void *beta, void *c, const MKL_INT ldc);
```

Include Files

- mkl.h

Description

The ?syrk routines perform a rank-k matrix-matrix operation for a symmetric matrix \( C \) using a general
matrix \( A \). The operation is defined as:

\[
C := \alpha A A' + \beta C,
\]

or

\[
C := \alpha A' A + \beta C,
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( C \) is an \( n \)-by-\( n \) symmetric matrix,
- \( A \) is an \( n \)-by-\( k \) matrix in the first case and a \( k \)-by-\( n \) matrix in the second case.
Input Parameters

**Layout**
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**
Specifies whether the upper or lower triangular part of the array \( c \) is used.
If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the array \( c \) is used.
If \( \text{uplo} = \text{CblasLower} \), then the lower triangular part of the array \( c \) is used.

**trans**
Specifies the operation:
if \( \text{trans} = \text{CblasNoTrans} \), then \( C := \alpha A A' + \beta C \);
if \( \text{trans} = \text{CblasTrans} \), then \( C := \alpha A'^* A + \beta C \);
if \( \text{trans} = \text{CblasConjTrans} \), then \( C := \alpha A'^* A + \beta C \).

**n**
Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

**k**
On entry with \( \text{trans} = \text{CblasNoTrans} \), \( k \) specifies the number of columns of the matrix \( A \), and on entry with \( \text{trans} = \text{CblasTrans} \) or \( \text{trans} = \text{CblasConjTrans} \), \( k \) specifies the number of rows of the matrix \( A \).
The value of \( k \) must be at least zero.

**alpha**
Specifies the scalar \( \alpha \).

**a**
Array, size \( \text{lda}*k \), where \( k \) is \( k \) when \( \text{trans} = \text{CblasNoTrans} \), and is \( n \) otherwise. Before entry with \( \text{trans} = \text{CblasNoTrans} \), the leading \( n \)-by-\( k \) part of the array \( a \) must contain the matrix \( A \), otherwise the leading \( k \)-by-\( n \) part of the array \( a \) must contain the matrix \( A \).

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>Layout = CblasRowMajor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array, size ( \text{lda}*k ). Before entry, the leading ( n )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
<td>Array, size ( \text{lda}*n ). Before entry, the leading ( k )-by-( n ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
</tbody>
</table>

**lda**

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>Layout = CblasRowMajor</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lda} ) must be at least ( \max(1, n) ).</td>
<td>( \text{lda} ) must be at least ( \max(1, k) ).</td>
</tr>
<tr>
<td>( \text{lda} ) must be at least ( \max(1, k) ).</td>
<td>( \text{lda} ) must be at least ( \max(1, n) ).</td>
</tr>
</tbody>
</table>
\( \beta \)

Specifies the scalar \( \beta \).

\( c \)

Array, size \( \text{ldc} \times n \). Before entry with \( \text{uplo} = \text{CblasUpper} \), the leading \( n \)-by-\( n \) upper triangular part of the array \( c \) must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of \( c \) is not referenced.

Before entry with \( \text{uplo} = \text{CblasLower} \), the leading \( n \)-by-\( n \) lower triangular part of the array \( c \) must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of \( c \) is not referenced.

\( \text{ldc} \)

Specifies the leading dimension of \( c \) as declared in the calling (sub)program. The value of \( \text{ldc} \) must be at least \( \max(1, n) \).

**Output Parameters**

\( c \)

With \( \text{uplo} = \text{CblasUpper} \), the upper triangular part of the array \( c \) is overwritten by the upper triangular part of the updated matrix.

With \( \text{uplo} = \text{CblasLower} \), the lower triangular part of the array \( c \) is overwritten by the lower triangular part of the updated matrix.

**cblas_\textcolor{red}{?}sy\textcolor{red}{r}2k**

**Performs a symmetric rank-2k update.**

**Syntax**

```c
void cblas_ssyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const float alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT ldb, const float beta, float *c, const MKL_INT ldc);
void cblas_dsyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const double alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT ldb, const double beta, double *c, const MKL_INT ldc);
void cblas_csyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
void cblas_zsyr2k (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE trans, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

**Include Files**

- mkl.h

**Description**

The \( \textcolor{red}{?}\textcolor{red}{sy}2k \) routines perform a rank-2k matrix-matrix operation for a symmetric matrix \( C \) using general matrices \( A \) and \( B \). The operation is defined as:

\[
C := \alpha A B' + \alpha B' A' + \beta C,
\]
or
\[ C := \alpha A' B + \alpha B' A + \beta C, \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( C \) is an \( n \times n \) symmetric matrix,
- \( A \) and \( B \) are \( n \times k \) matrices in the first case, and \( k \times n \) matrices in the second case.

**Input Parameters**

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**uplo**

Specifies whether the upper or lower triangular part of the array \( c \) is used.

- If \( \text{uplo} = \text{CblasUpper} \), then the upper triangular part of the array \( c \) is used.
- If \( \text{uplo} = \text{CblasLower} \), then the lower triangular part of the array \( c \) is used.

**trans**

Specifies the operation:

- If \( \text{trans} = \text{CblasNoTrans} \), then \( C := \alpha A' B + \alpha B' A + \beta C \);
- If \( \text{trans} = \text{CblasTrans} \), then \( C := \alpha A' B + \alpha B' A + \beta C \);
- If \( \text{trans} = \text{CblasConjTrans} \), then \( C := \alpha A' B + \alpha B' A + \beta C \).

**n**

Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

**k**

On entry with \( \text{trans} = \text{CblasNoTrans} \), \( k \) specifies the number of columns of the matrices \( A \) and \( B \), and on entry with \( \text{trans} = \text{CblasTrans} \) or \( \text{trans} = \text{CblasConjTrans} \), \( k \) specifies the number of rows of the matrices \( A \) and \( B \). The value of \( k \) must be at least zero.

**alpha**

Specifies the scalar \( \alpha \).

**a**

Specifies the two-dimensional array \( a \) as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array, size lda*k</td>
<td>Before entry, the leading ( n )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
<td>Array, size lda*n. Before entry, the leading ( k )-by-( n ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
<th>trans=CblasNoTrans</th>
<th>trans=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array, size lda*n</td>
<td>Before entry, the leading ( k )-by-( n ) part of the array ( a ) must contain the matrix ( A ).</td>
<td>Array, size lda*k. Before entry, the leading ( n )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
</tbody>
</table>

**lda**

Specifies the leading dimension of \( a \) as declared in the calling (sub)program.
trans = CblasNoTrans, trans = CblasConjTrans

### Layout = CblasColMajor

<table>
<thead>
<tr>
<th></th>
<th>trans = CblasNoTrans</th>
<th>trans = CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda</td>
<td>must be at least (\max(1, n)).</td>
<td>must be at least (\max(1, k)).</td>
</tr>
</tbody>
</table>

### Layout = CblasRowMajor

<table>
<thead>
<tr>
<th></th>
<th>trans = CblasNoTrans</th>
<th>trans = CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda</td>
<td>must be at least (\max(1, k)).</td>
<td>must be at least (\max(1, n)).</td>
</tr>
</tbody>
</table>

\(b\) trans = CblasNoTrans, trans = CblasConjTrans

### Layout = CblasColMajor

<table>
<thead>
<tr>
<th></th>
<th>Array, size (ldb \times k).</th>
<th>Array, size (ldb \times n).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before entry, the leading (n)-by-(k) part of the array (b) must contain the matrix (B).</td>
<td>Before entry, the leading (k)-by-(n) part of the array (b) must contain the matrix (B).</td>
</tr>
</tbody>
</table>

### Layout = CblasRowMajor

<table>
<thead>
<tr>
<th></th>
<th>Array, size (lda \times n).</th>
<th>Array, size (lda \times k).</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Before entry, the leading (k)-by-(n) part of the array (b) must contain the matrix (B).</td>
<td>Before entry, the leading (n)-by-(k) part of the array (b) must contain the matrix (B).</td>
</tr>
</tbody>
</table>

\(ldb\) Specifies the leading dimension of \(a\) as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th></th>
<th>trans = CblasNoTrans</th>
<th>trans = CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lda must be at least (\max(1, n)).</td>
<td>lda must be at least (\max(1, k)).</td>
</tr>
</tbody>
</table>

### beta

Specifies the scalar \(beta\).

\(c\) Array, size \(ldc \times n\). Before entry with \(uplo = \text{CblasUpper}\), the leading \(n\)-by-\(n\) upper triangular part of the array \(c\) must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of \(c\) is not referenced.

Before entry with \(uplo = \text{CblasLower}\), the leading \(n\)-by-\(n\) lower triangular part of the array \(c\) must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of \(c\) is not referenced.

\(ldc\) Specifies the leading dimension of \(c\) as declared in the calling (sub)program. The value of \(ldc\) must be at least \(\max(1, n)\).

### Output Parameters

\(c\) With \(uplo = \text{CblasUpper}\), the upper triangular part of the array \(c\) is overwritten by the upper triangular part of the updated matrix.
With \( \text{uplo} = \text{CblasLower} \), the lower triangular part of the array \( c \) is overwritten by the lower triangular part of the updated matrix.

**cblas_?trmm**  
Computes a matrix-matrix product where one input matrix is triangular.

**Syntax**

void cblas_strmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const MKL_INT ldb);

void cblas_dtrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const MKL_INT ldb);

void cblas_ctrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);

void cblas_ztrmm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);

**Include Files**

- mkl.h

**Description**

The \( ? \text{trmm} \) routines compute a scalar-matrix-matrix product with one triangular matrix. The operation is defined as

\[
B := \alpha \cdot \text{op}(A) \cdot B
\]

or

\[
B := \alpha \cdot B \cdot \text{op}(A)
\]

where:

- \( \alpha \) is a scalar,
- \( B \) is an \( m \)-by-\( n \) matrix,
- \( A \) is a unit, or non-unit, upper or lower triangular matrix

\( \text{op}(A) \) is one of \( \text{op}(A) = A \), \( \text{op}(A) = A' \), or \( \text{op}(A) = \text{conjg}(A') \).

**Input Parameters**

- **Layout** Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).
- **side** Specifies whether \( \text{op}(A) \) appears on the left or right of \( B \) in the operation:
if $side = \text{CblasLeft}$, then $B := \alpha \cdot \text{op}(A) \cdot B$;
if $side = \text{CblasRight}$, then $B := \alpha \cdot B \cdot \text{op}(A)$.

$\text{uplo}$
Specifies whether the matrix $A$ is upper or lower triangular.

$\text{uplo} = \text{CblasUpper}$
if $\text{uplo} = \text{CblasLower}$, then the matrix is lower triangular.

$\text{transa}$
Specifies the form of $\text{op}(A)$ used in the matrix multiplication:
if $\text{transa}=\text{CblasNoTrans}$, then $\text{op}(A) = A$;
if $\text{transa}=\text{CblasTrans}$, then $\text{op}(A) = A'$;
if $\text{transa}=\text{CblasConjTrans}$, then $\text{op}(A) = \text{conjg}(A')$.

$\text{diag}$
Specifies whether the matrix $A$ is unit triangular:
if $\text{diag} = \text{CblasUnit}$ then the matrix is unit triangular;
if $\text{diag} = \text{CblasNonUnit}$, then the matrix is not unit triangular.

$m$
Specifies the number of rows of $B$. The value of $m$ must be at least zero.

$n$
Specifies the number of columns of $B$. The value of $n$ must be at least zero.

$\alpha$
Specifies the scalar $\alpha$.
When $\alpha$ is zero, then $a$ is not referenced and $b$ need not be set before entry.

$a$
Array, size $lda$ by $k$, where $k$ is $m$ when $side = \text{CblasLeft}$ and is $n$ when $side = \text{CblasRight}$. Before entry with $\text{uplo} = \text{CblasUpper}$, the leading $k$ by $k$ upper triangular part of the array $a$ must contain the upper triangular matrix and the strictly lower triangular part of $a$ is not referenced.
Before entry with $\text{uplo} = \text{CblasLower}$, the leading $k$ by $k$ lower triangular part of the array $a$ must contain the lower triangular matrix and the strictly upper triangular part of $a$ is not referenced.
When $\text{diag} = \text{CblasUnit}$, the diagonal elements of $a$ are not referenced either, but are assumed to be unity.

$lda$
Specifies the leading dimension of $a$ as declared in the calling (sub)program. When $side = \text{CblasLeft}$, then $lda$ must be at least $\max(1, m)$, when $side = \text{CblasRight}$, then $lda$ must be at least $\max(1, n)$.

$b$
For Layout = CblasColMajor: array, size $ldb \cdot n$. Before entry, the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.
For Layout = CblasRowMajor: array, size $ldb \cdot m$. Before entry, the leading $n$-by-$m$ part of the array $b$ must contain the matrix $B$.

$ldb$
Specifies the leading dimension of $b$ as declared in the calling (sub)program. When Layout = CblasColMajor, $ldb$ must be at least $\max(1, m)$; otherwise, $ldb$ must be at least $\max(1, n)$.

Output Parameters

$b$
Overwritten by the transformed matrix.
cblas_?trsm
Solves a triangular matrix equation.

Syntax
void cblas_strsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const float alpha, const float *a, const MKL_INT lda, float *b, const MKL_INT ldb);
void cblas_dtrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const double alpha, const double *a, const MKL_INT lda, double *b, const MKL_INT ldb);
void cblas_ctrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);
void cblas_ztrsm (const CBLAS_LAYOUT Layout, const CBLAS_SIDE side, const CBLAS_UPLO uplo, const CBLAS_TRANSPOSE transa, const CBLAS_DIAG diag, const MKL_INT m, const MKL_INT n, const void *alpha, const void *a, const MKL_INT lda, void *b, const MKL_INT ldb);

Include Files
• mkl.h

Description
The ?trsm routines solve one of the following matrix equations:

\[ \text{op}(A) \cdot X = \alpha \cdot B, \]

or

\[ X \cdot \text{op}(A) = \alpha \cdot B, \]

where:
\( \alpha \) is a scalar,
\( X \) and \( B \) are \( m \)-by-\( n \) matrices,
\( A \) is a unit, or non-unit, upper or lower triangular matrix, and
\( \text{op}(A) \) is one of \( \text{op}(A) = A, \text{op}(A) = A', \text{or} \text{op}(A) = \text{conjg}(A') \).

The matrix \( B \) is overwritten by the solution matrix \( X \).

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **side**
  Specifies whether \( \text{op}(A) \) appears on the left or right of \( X \) in the equation:
  if \( \text{side} = \text{CblasLeft} \), then \( \text{op}(A) \cdot X = \alpha \cdot B; \)
  if \( \text{side} = \text{CblasRight} \), then \( X \cdot \text{op}(A) = \alpha \cdot B. \)
Specifies whether the matrix $A$ is upper or lower triangular.

$uplo = \text{CblasUpper}$

if $uplo = \text{CblasLower}$, then the matrix is lower triangular.

Specifies the form of $\text{op}(A)$ used in the matrix multiplication:

if $transa = \text{CblasNoTrans}$, then $\text{op}(A) = A$;

if $transa = \text{CblasTrans}$;

if $transa = \text{CblasConjTrans}$, then $\text{op}(A) = \text{conjg}(A')$.

Specifies whether the matrix $A$ is unit triangular:

if $diag = \text{CblasUnit}$ then the matrix is unit triangular;

if $diag = \text{CblasNonUnit}$, then the matrix is not unit triangular.

Specifies the number of rows of $B$. The value of $m$ must be at least zero.

Specifies the number of columns of $B$. The value of $n$ must be at least zero.

Specifies the scalar $\alpha$.

When $\alpha$ is zero, then $a$ is not referenced and $b$ need not be set before entry.

Array, size $lda \times k$, where $k$ is $m$ when $side = \text{CblasLeft}$ and is $n$ when $side = \text{CblasRight}$. Before entry with $uplo = \text{CblasUpper}$, the leading $k$ by $k$ upper triangular part of the array $a$ must contain the upper triangular matrix and the strictly lower triangular part of $a$ is not referenced.

Before entry with $uplo = \text{CblasLower}$ lower triangular part of the array $a$ must contain the lower triangular matrix and the strictly upper triangular part of $a$ is not referenced.

When $diag = \text{CblasUnit}$, the diagonal elements of $a$ are not referenced either, but are assumed to be unity.

Specifies the leading dimension of $a$ as declared in the calling (sub)program. When $side = \text{CblasLeft}$, then $lda$ must be at least $\max(1, m)$, when $side = \text{CblasRight}$, then $lda$ must be at least $\max(1, n)$.

For $Layout = \text{CblasColMajor}$: array, size $ldb \times n$. Before entry, the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

For $Layout = \text{CblasRowMajor}$: array, size $ldb \times m$. Before entry, the leading $n$-by-$m$ part of the array $b$ must contain the matrix $B$.

Specifies the leading dimension of $b$ as declared in the calling (sub)program. When $Layout = \text{CblasColMajor}$, $ldb$ must be at least $\max(1, m)$; otherwise, $ldb$ must be at least $\max(1, n)$.

**Output Parameters**

Overwritten by the solution matrix $X$. 

---

Intel® Math Kernel Library Developer Reference
Sparse BLAS Level 1 Routines

This section describes Sparse BLAS Level 1, an extension of BLAS Level 1 included in the Intel® Math Kernel Library beginning with the Intel® MKL release 2.1. Sparse BLAS Level 1 is a group of routines and functions that perform a number of common vector operations on sparse vectors stored in compressed form.

Sparse vectors are those in which the majority of elements are zeros. Sparse BLAS routines and functions are specially implemented to take advantage of vector sparsity. This allows you to achieve large savings in computer time and memory. If nz is the number of non-zero vector elements, the computer time taken by Sparse BLAS operations will be $O(nz)$.

Vector Arguments

Compressed sparse vectors. Let $a$ be a vector stored in an array, and assume that the only non-zero elements of $a$ are the following:

$$a[k_1], a[k_2], a[k_3] \ldots a[k_{nz}],$$

where $nz$ is the total number of non-zero elements in $a$.

In Sparse BLAS, this vector can be represented in compressed form by two arrays, $x$ (values) and $indx$ (indices). Each array has $nz$ elements:

$$x[0]=a[k_1], x[1]=a[k_2], \ldots x[nz-1]=a[k_{nz}],$$

$$indx[0]=k_1, indx[1]=k_2, \ldots indx[nz-1]=k_{nz}.$$ 

Thus, a sparse vector is fully determined by the triple ($nz$, $x$, $indx$). If you pass a negative or zero value of $nz$ to Sparse BLAS, the subroutines do not modify any arrays or variables.

Full-storage vectors. Sparse BLAS routines can also use a vector argument fully stored in a single array (a full-storage vector). If $y$ is a full-storage vector, its elements must be stored contiguously: the first element in $y[0]$, the second in $y[1]$, and so on. This corresponds to an increment $incy = 1$ in BLAS Level 1. No increment value for full-storage vectors is passed as an argument to Sparse BLAS routines or functions.

Naming Conventions for Sparse BLAS Routines

Similar to BLAS, the names of Sparse BLAS subprograms have prefixes that determine the data type involved: $s$ and $d$ for single- and double-precision real; $c$ and $z$ for single- and double-precision complex respectively.

If a Sparse BLAS routine is an extension of a "dense" one, the subprogram name is formed by appending the suffix $i$ (standing for indexed) to the name of the corresponding "dense" subprogram. For example, the Sparse BLAS routine $saxpyi$ corresponds to the BLAS routine $saxpy$, and the Sparse BLAS function $cdotci$ corresponds to the BLAS function $cdotc$.

Routines and Data Types

Routines and data types supported in the Intel® MKL implementation of Sparse BLAS are listed in Table “Sparse BLAS Routines and Their Data Types”.

<table>
<thead>
<tr>
<th>Routine/Function</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblas_?axpyi</td>
<td>s, d, c, z</td>
<td>Scalar-vector product plus vector (routines)</td>
</tr>
<tr>
<td>cblas_?doti</td>
<td>s, d</td>
<td>Dot product (functions)</td>
</tr>
<tr>
<td>cblas_?dotci</td>
<td>c, z</td>
<td>Complex dot product conjugated (functions)</td>
</tr>
<tr>
<td>Routine/Function</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>cblas_?dotui</td>
<td>c, z</td>
<td>Complex dot product unconjugated (functions)</td>
</tr>
<tr>
<td>cblas_?gthr</td>
<td>s, d, c, z</td>
<td>Gathering a full-storage sparse vector into compressed form (nz, x, \text{indx} ) (routines)</td>
</tr>
<tr>
<td>cblas_?gthrz</td>
<td>s, d, c, z</td>
<td>Gathering a full-storage sparse vector into compressed form and assigning zeros to gathered elements in the full-storage vector (routines)</td>
</tr>
<tr>
<td>cblas_?roti</td>
<td>s, d</td>
<td>Givens rotation (routines)</td>
</tr>
<tr>
<td>cblas_?sctr</td>
<td>s, d, c, z</td>
<td>Scattering a vector from compressed form to full-storage form (routines)</td>
</tr>
</tbody>
</table>

BLAS Level 1 Routines That Can Work With Sparse Vectors

The following BLAS Level 1 routines will give correct results when you pass to them a compressed-form array \(x\) (with the increment \(\text{incx}=1\)):

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblas_?asum</td>
<td>sum of absolute values of vector elements</td>
</tr>
<tr>
<td>cblas_?copy</td>
<td>copying a vector</td>
</tr>
<tr>
<td>cblas_?nrm2</td>
<td>Euclidean norm of a vector</td>
</tr>
<tr>
<td>cblas_?scal</td>
<td>scaling a vector</td>
</tr>
<tr>
<td>cblas_i?amax</td>
<td>index of the element with the largest absolute value for real flavors, or the largest sum (</td>
</tr>
<tr>
<td>cblas_i?amin</td>
<td>index of the element with the smallest absolute value for real flavors, or the smallest sum (</td>
</tr>
</tbody>
</table>

The result \(i\) returned by \(i?amax\) and \(i?amin\) should be interpreted as index in the compressed-form array, so that the largest (smallest) value is \(x[i-1]\); the corresponding index in full-storage array is \(\text{indx}[i-1]\).

You can also call \(\text{cblas}_?\text{rotg}\) to compute the parameters of Givens rotation and then pass these parameters to the Sparse BLAS routines \(\text{cblas}_?\text{roti}\).

cblas_?axpyi

_Adds a scalar multiple of compressed sparse vector to a full-storage vector._

Syntax

```c
void cblas_saxpyi (const MKL_INT nz, const float a, const float *x, const MKL_INT *indx, float *y);
void cblas_daxpyi (const MKL_INT nz, const double a, const double *x, const MKL_INT *indx, double *y);
void cblas_caxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx, void *y);
void cblas_zaxpyi (const MKL_INT nz, const void *a, const void *x, const MKL_INT *indx, void *y);
```
Include Files

- mkl.h

Description

The `?axpyi` routines perform a vector-vector operation defined as

\[ y := a \cdot x + y \]

where:
- \( a \) is a scalar,
- \( x \) is a sparse vector stored in compressed form,
- \( y \) is a vector in full storage form.

The `?axpyi` routines reference or modify only the elements of \( y \) whose indices are listed in the array \( indx \).

The values in \( indx \) must be distinct.

Input Parameters

- \( nz \) The number of elements in \( x \) and \( indx \).
- \( a \) Specifies the scalar \( a \).
- \( x \) Array, size at least \( nz \).
- \( indx \) Specifies the indices for the elements of \( x \).
  Array, size at least \( nz \).
- \( y \) Array, size at least \( \max(indx[i]) \).

Output Parameters

- \( y \) Contains the updated vector \( y \).

**cblas_?doti**

*Computes the dot product of a compressed sparse real vector by a full-storage real vector.*

Syntax

```c
float cblas_sdoti (const MKL_INT nz, const float *x, const MKL_INT *indx, const float *y);
double cblas_ddoti (const MKL_INT nz, const double *x, const MKL_INT *indx, const double *y);
```

Include Files

- mkl.h

Description

The `?doti` routines return the dot product of \( x \) and \( y \) defined as

\[ \text{res} = x[0]y[indx[0]] + x[1]y[indx[1]] + \ldots + x[nz-1]y[indx[nz-1]] \]
where the triple \((nz, x, indx)\) defines a sparse real vector stored in compressed form, and \(y\) is a real vector in full storage form. The functions reference only the elements of \(y\) whose indices are listed in the array \(indx\). The values in \(indx\) must be distinct.

**Input Parameters**

- \(nz\) The number of elements in \(x\) and \(indx\).
- \(x\) Array, size at least \(nz\).
- \(indx\) Specifies the indices for the elements of \(x\). Array, size at least \(nz\).
- \(y\) Array, size at least \(\max(indx[i])\).

**Output Parameters**

- \(res\) Contains the dot product of \(x\) and \(y\), if \(nz\) is positive. Otherwise, \(res\) contains 0.

**cblas_?dotci**

*Computes the conjugated dot product of a compressed sparse complex vector with a full-storage complex vector.*

**Syntax**

```c
void cblas_cdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
void cblas_zdotci_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
```

**Include Files**

- mkl.h

**Description**

The ?dotci routines return the dot product of \(x\) and \(y\) defined as

\[
\text{conjg}(x[0])y[indx[0]] + \ldots + \text{conjg}(x[nz-1])y[indx[nz-1]]
\]

where the triple \((nz, x, indx)\) defines a sparse complex vector stored in compressed form, and \(y\) is a real vector in full storage form. The functions reference only the elements of \(y\) whose indices are listed in the array \(indx\). The values in \(indx\) must be distinct.

**Input Parameters**

- \(nz\) The number of elements in \(x\) and \(indx\).
- \(x\) Array, size at least \(nz\).
- \(indx\) Specifies the indices for the elements of \(x\). Array, size at least \(nz\).
- \(y\) Array, size at least \(\max(indx[i])\).
Output Parameters

\textit{dotui} 

Contains the conjugated dot product of \textit{x} and \textit{y}, if \textit{nz} is positive. Otherwise, it contains 0.

\textbf{cblas\textunderscore ?dotui}

\textit{Computes the dot product of a compressed sparse complex vector by a full-storage complex vector.}

\textbf{Syntax}

\begin{verbatim}
void cblas_cdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
void cblas_zdotui_sub (const MKL_INT nz, const void *x, const MKL_INT *indx, const void *y, void *dotui);
\end{verbatim}

\textbf{Include Files}

- mkl.h

\textbf{Description}

The \textit{?dotui} routines return the dot product of \textit{x} and \textit{y} defined as

\[ \text{res} = x[0] \times y[\text{indx}[0]] + x[1] \times y(\text{indx}[1]) + \ldots + x[nz - 1] \times y(\text{indx}[nz - 1]) \]

where the triple (\textit{nz}, \textit{x}, \textit{indx}) defines a sparse complex vector stored in compressed form, and \textit{y} is a real vector in full storage form. The functions reference only the elements of \textit{y} whose indices are listed in the array \textit{indx}. The values in \textit{indx} must be distinct.

\textbf{Input Parameters}

\begin{itemize}
  \item \textit{nz} 
    The number of elements in \textit{x} and \textit{indx}.
  \item \textit{x} 
    Array, size at least \textit{nz}.
  \item \textit{indx} 
    Specifies the indices for the elements of \textit{x}.
    Array, size at least \textit{nz}.
  \item \textit{y} 
    Array, size at least max(\textit{indx}[j]).
\end{itemize}

\textbf{Output Parameters}

\begin{itemize}
  \item \textit{dotui} 
    Contains the dot product of \textit{x} and \textit{y}, if \textit{nz} is positive. Otherwise, \textit{res} contains 0.
\end{itemize}

\textbf{cblas\textunderscore ?gthr}

\textit{Gathers a full-storage sparse vector’s elements into compressed form.}

\textbf{Syntax}

\begin{verbatim}
void cblas_sgthr (const MKL_INT nz, const float *y, float *x, const MKL_INT *indx);
void cblas_dgthr (const MKL_INT nz, const double *y, double *x, const MKL_INT *indx);
\end{verbatim}
void cblas_cgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);
void cblas_zgthr (const MKL_INT nz, const void *y, void *x, const MKL_INT *indx);

Include Files
• mkl.h

Description
The ?gthr routines gather the specified elements of a full-storage sparse vector y into compressed form (nz, x, indx). The routines reference only the elements of y whose indices are listed in the array indx:
\[ x[i] = y[indx[i]], \text{ for } i=0,1,\ldots,nz-1. \]

Input Parameters
\begin{itemize}
  \item \textit{nz} \hfill The number of elements of \textit{y} to be gathered.
  \item \textit{indx} \hfill Specifies indices of elements to be gathered.
    \begin{itemize}
      \item Array, size at least \textit{nz}.
    \end{itemize}
  \item \textit{y} \hfill Array, size at least \text{max(indx[i])}.
\end{itemize}

Output Parameters
\begin{itemize}
  \item \textit{x} \hfill Array, size at least \textit{nz}.
    \begin{itemize}
      \item Contains the vector converted to the compressed form.
    \end{itemize}
\end{itemize}

cblas_?gthrz
\textit{Gathers a sparse vector's elements into compressed form, replacing them by zeros.}

Syntax
void cblas_sgthrz (const MKL_INT nz, float *y, float *x, const MKL_INT *indx);
void cblas_dgthrz (const MKL_INT nz, double *y, double *x, const MKL_INT *indx);
void cblas_cgthrz (const MKL_INT nz, void *y, void *x, const MKL_INT *indx);
void cblas_zgthrz (const MKL_INT nz, void *y, void *x, const MKL_INT *indx);

Include Files
• mkl.h

Description
The ?gthrz routines gather the elements with indices specified by the array \textit{indx} from a full-storage vector \textit{y} into compressed form (\textit{nz}, \textit{x}, \textit{indx}) and overwrite the gathered elements of \textit{y} by zeros. Other elements of \textit{y} are not referenced or modified (see also ?gthr).

Input Parameters
\begin{itemize}
  \item \textit{nz} \hfill The number of elements of \textit{y} to be gathered.
  \item \textit{indx} \hfill Specifies indices of elements to be gathered.
\end{itemize}
Array, size at least \( nz \).

\( y \)

Array, size at least \( \max(\text{indx}[i]) \).

**Output Parameters**

\( x \)

Array, size at least \( nz \).

Contains the vector converted to the compressed form.

\( y \)

The updated vector \( y \).

**cblas_?roti**

Applies Givens rotation to sparse vectors one of which is in compressed form.

**Syntax**

```c
void cblas_sroti (const MKL_INT \( nz \), float *x, const MKL_INT *indx, float *y, const float c, const float s);
void cblas_droti (const MKL_INT \( nz \), double *x, const MKL_INT *indx, double *y, const double c, const double s);
```

**Include Files**

- mkl.h

**Description**

The \(?\text{roti}\) routines apply the Givens rotation to elements of two real vectors, \( x \) (in compressed form \( nz, x, \text{indx} \)) and \( y \) (in full storage form):

\[
\begin{align*}
    x[i] &= c*x[i] + s*y[\text{indx}[i]] \\
    y[\text{indx}[i]] &= c*y[\text{indx}[i]] - s*x[i]
\end{align*}
\]

The routines reference only the elements of \( y \) whose indices are listed in the array \( \text{indx} \). The values in \( \text{indx} \) must be distinct.

**Input Parameters**

- \( nz \)
  - The number of elements in \( x \) and \( \text{indx} \).
- \( x \)
  - Array, size at least \( nz \).
- \( \text{indx} \)
  - Specifies the indices for the elements of \( x \).
  - Array, size at least \( nz \).
- \( y \)
  - Array, size at least \( \max(\text{indx}[i]) \).
- \( c \)
  - A scalar.
- \( s \)
  - A scalar.

**Output Parameters**

- \( x \) and \( y \)
  - The updated arrays.
cblas_?sctr
Converts compressed sparse vectors into full storage form.

Syntax
void cblas_ssctr (const MKL_INT nz, const float *x, const MKL_INT *indx, float *y);
void cblas_dsctr (const MKL_INT nz, const double *x, const MKL_INT *indx, double *y);
void cblas_csctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);
void cblas_zsctr (const MKL_INT nz, const void *x, const MKL_INT *indx, void *y);

Include Files
• mkl.h

Description
The ?sctr routines scatter the elements of the compressed sparse vector \((nz, x, indx)\) to a full-storage vector \(y\). The routines modify only the elements of \(y\) whose indices are listed in the array \(indx\):
\[ y[indx[i]] = x[i], \text{for } i=0,1,...,nz-1. \]

Input Parameters
\(nz\) The number of elements of \(x\) to be scattered.
\(indx\) Specifies indices of elements to be scattered.
Array, size at least \(nz\).
\(x\) Array, size at least \(nz\).
Contains the vector to be converted to full-storage form.

Output Parameters
\(y\) Array, size at least \(\max(indx[i])\).
Contains the vector \(y\) with updated elements.

Sparse BLAS Level 2 and Level 3 Routines

NOTE The Intel® MKL Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® MKL Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

This section describes Sparse BLAS Level 2 and Level 3 routines included in the Intel® Math Kernel Library (Intel® MKL). Sparse BLAS Level 2 is a group of routines and functions that perform operations between a sparse matrix and dense vectors. Sparse BLAS Level 3 is a group of routines and functions that perform operations between a sparse matrix and dense matrices.

The terms and concepts required to understand the use of the Intel® MKL Sparse BLAS Level 2 and Level 3 routines are discussed in the Linear Solvers Basics appendix.
The Sparse BLAS routines can be useful to implement iterative methods for solving large sparse systems of equations or eigenvalue problems. For example, these routines can be considered as building blocks for Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS) described in the Chapter 8 of the manual.

Intel® MKL provides Sparse BLAS Level 2 and Level 3 routines with typical (or conventional) interface similar to the interface used in the NIST* Sparse BLAS library [Rem05].

Some software packages and libraries (the PARDISO* Solver used in Intel® MKL, Sparskit 2 [Saad94], the Compaq* Extended Math Library (CXML)[CXML01]) use different (early) variation of the compressed sparse row (CSR) format and support only Level 2 operations with simplified interfaces. Intel® MKL provides an additional set of Sparse BLAS Level 2 routines with similar simplified interfaces. Each of these routines operates only on a matrix of the fixed type.

The routines described in this section support both one-based indexing and zero-based indexing of the input data (see details in the section One-based and Zero-based Indexing).

### Naming Conventions in Sparse BLAS Level 2 and Level 3

Each Sparse BLAS Level 2 and Level 3 routine has a six- or eight-character base name preceded by the prefix mkl_ or mkl_cspblas_.

The routines with typical (conventional) interface have six-character base names in accordance with the template:

```
mkl_<character> <data> <operation>( )
```

The routines with simplified interfaces have eight-character base names in accordance with the templates:

```
mkl_<character> <data> <mtype> <operation>( )
```

for routines with one-based indexing; and

```
mkl_cspblas_<character> <data> <mtype> <operation>( )
```

for routines with zero-based indexing.

The <character> field indicates the data type:

- **s**: real, single precision
- **c**: complex, single precision
- **d**: real, double precision
- **z**: complex, double precision

The <data> field indicates the sparse matrix storage format (see section Sparse Matrix Storage Formats):

- **coo**: coordinate format
- **csr**: compressed sparse row format and its variations
- **csc**: compressed sparse column format and its variations
- **dia**: diagonal format
- **sky**: skyline storage format
- **bsr**: block sparse row format and its variations

The <operation> field indicates the type of operation:

- **mv**: matrix-vector product (Level 2)
- **mm**: matrix-matrix product (Level 3)
- **sv**: solving a single triangular system (Level 2)
The field `<mtype>` indicates the matrix type:
- `ge` sparse representation of a general matrix
- `sy` sparse representation of the upper or lower triangle of a symmetric matrix
- `tr` sparse representation of a triangular matrix

**Sparse Matrix Storage Formats for Sparse BLAS Routines**

The current version of Intel® MKL Sparse BLAS Level 2 and Level 3 routines support the following point entry [Duff86] storage formats for sparse matrices:
- *compressed sparse row* format (CSR) and its variations;
- *compressed sparse column* format (CSC);
- *coordinate* format;
- *diagonal* format;
- *skyline* storage format;

and one block entry storage format:
- *block sparse row* format (BSR) and its variations.

For more information see "Sparse Matrix Storage Formats" in Appendix A.

Intel® MKL provides auxiliary routines -matrix converters - that convert sparse matrix from one storage format to another.

**Routines and Supported Operations**

This section describes operations supported by the Intel® MKL Sparse BLAS Level 2 and Level 3 routines. The following notations are used here:
- \(A\) is a sparse matrix;
- \(B\) and \(C\) are dense matrices;
- \(D\) is a diagonal scaling matrix;
- \(x\) and \(y\) are dense vectors;
- \(\alpha\) and \(\beta\) are scalars;

\(\text{op}(A)\) is one of the possible operations:
- \(\text{op}(A) = A\);
- \(\text{op}(A) = A^T\) - transpose of \(A\);
- \(\text{op}(A) = A^H\) - conjugated transpose of \(A\).

\(\text{inv}(\text{op}(A))\) denotes the inverse of \(\text{op}(A)\).

The Intel® MKL Sparse BLAS Level 2 and Level 3 routines support the following operations:
- computing the vector product between a sparse matrix and a dense vector:
  \(y := \alpha \text{op}(A) x + \beta y\)
- solving a single triangular system:
  \(y := \alpha \text{inv}(\text{op}(A)) x\)
- computing a product between sparse matrix and dense matrix:
  \(C := \alpha \text{op}(A) B + \beta C\)
- solving a sparse triangular system with multiple right-hand sides:
  \(C := \alpha \text{inv}(\text{op}(A)) B\)
Intel® MKL provides an additional set of the Sparse BLAS Level 2 routines with simplified interfaces. Each of these routines operates on a matrix of the fixed type. The following operations are supported:

- computing the vector product between a sparse matrix and a dense vector (for general and symmetric matrices):
  \[ y := \text{op}(A) \cdot x \]
- solving a single triangular system (for triangular matrices):
  \[ y := \text{inv(op}(A) \cdot x \]

Matrix type is indicated by the field \(<mtype>\) in the routine name (see section Naming Conventions in Sparse BLAS Level 2 and Level 3).

**NOTE**
The routines with simplified interfaces support only four sparse matrix storage formats, specifically:
- CSR format in the 3-array variation accepted in the direct sparse solvers and in the CXML;
- diagonal format accepted in the CXML;
- coordinate format;
- BSR format in the 3-array variation.

Note that routines with both typical (conventional) and simplified interfaces use the same computational kernels that work with certain internal data structures.

The Intel® MKL Sparse BLAS Level 2 and Level 3 routines do not support in-place operations.

Complete list of all routines is given in the “Sparse BLAS Level 2 and Level 3 Routines”.

### Interface Consideration

#### One-Based and Zero-Based Indexing
The Intel® MKL Sparse BLAS Level 2 and Level 3 routines support one-based and zero-based indexing of data arrays.

Routines with typical interfaces support zero-based indexing for the following sparse data storage formats: CSR, CSC, BSR, and COO. Routines with simplified interfaces support zero based indexing for the following sparse data storage formats: CSR, BSR, and COO. See the complete list of Sparse BLAS Level 2 and Level 3 Routines.

The one-based indexing uses the convention of starting array indices at 1. The zero-based indexing uses the convention of starting array indices at 0. For example, indices of the 5-element array \(x\) can be presented in case of one-based indexing as follows:

Element index: 1 2 3 4 5
Element value: 1.0 5.0 7.0 8.0 9.0

and in case of zero-based indexing as follows:

Element index: 0 1 2 3 4
Element value: 1.0 5.0 7.0 8.0 9.0

The detailed descriptions of the one-based and zero-based variants of the sparse data storage formats are given in the "Sparse Matrix Storage Formats" in Appendix A.

Most parameters of the routines are identical for both one-based and zero-based indexing, but some of them have certain differences. The following table lists all these differences.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>One-based Indexing</th>
<th>Zero-based Indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>val</strong></td>
<td>Array containing non-zero elements of the matrix ( A ), its length is ( \text{pntrb}[m] - \text{pntrb}[1] )</td>
<td>Array containing non-zero elements of the matrix ( A ), its length is ( \text{pntrb}[m-1] - \text{pntrb}[0] )</td>
</tr>
<tr>
<td><strong>pntrb</strong></td>
<td>Array of length ( m ). This array contains row indices, such that ( \text{pntrb}[i] - \text{pntrb}[i+1] ) is the first index of row ( i ) in the arrays ( \text{val} ) and ( \text{indx} )</td>
<td>Array of length ( m ). This array contains row indices, such that ( \text{pntrb}[i] - \text{pntrb}[0] ) is the first index of row ( i ) in the arrays ( \text{val} ) and ( \text{indx} )</td>
</tr>
<tr>
<td><strong>pntrc</strong></td>
<td>Array of length ( m ). This array contains row indices, such that ( \text{pntrc}[i] - \text{pntrc}[i+1] ) is the last index of row ( i ) in the arrays ( \text{val} ) and ( \text{indx} )</td>
<td>Array of length ( m ). This array contains row indices, such that ( \text{pntrc}[i] - \text{pntrc}[0] ) is the last index of row ( i ) in the arrays ( \text{val} ) and ( \text{indx} )</td>
</tr>
<tr>
<td><strong>ia</strong></td>
<td>Array of length ( m+1 ), containing indices of elements in the array ( A ), such that ( \text{ia}[i] ) is the index in the array ( A ) of the first non-zero element from the row ( i ). The value of the last element ( \text{ia}[m+1] ) is equal to the number of non-zeros plus one.</td>
<td>Array of length ( m+1 ), containing indices of elements in the array ( A ), such that ( \text{ia}[i] ) is the index in the array ( A ) of the first non-zero element from the row ( i ). The value of the last element ( \text{ia}[m] ) is equal to the number of non-zeros.</td>
</tr>
<tr>
<td><strong>ldb</strong></td>
<td>Specifies the leading dimension of ( B ) as declared in the calling (sub)program.</td>
<td>Specifies the second dimension of ( B ) as declared in the calling (sub)program.</td>
</tr>
<tr>
<td><strong>ldc</strong></td>
<td>Specifies the leading dimension of ( C ) as declared in the calling (sub)program.</td>
<td>Specifies the second dimension of ( C ) as declared in the calling (sub)program.</td>
</tr>
</tbody>
</table>

**Differences Between Intel MKL and NIST* Interfaces**

The Intel® MKL Sparse BLAS Level 3 routines have the following conventional interfaces:

\[
mkl_{\text{xyyym}}(\text{transa}, m, n, k, \alpha, \text{matdescra}, \text{arg}(A), b, \text{ldb}, \beta, c, \text{ldc}), \text{for matrix-matrix product};
\]

\[
mkl_{\text{xyyysm}}(\text{transa}, m, n, \alpha, \text{matdescra}, \text{arg}(A), b, \text{ldb}, c, \text{ldc}), \text{for triangular solvers with multiple right-hand sides}.
\]

Here \( x \) denotes data type, and \( yyy \) - sparse matrix data structure (storage format).

The analogous NIST* Sparse BLAS (NSB) library routines have the following interfaces:

\[
x_{\text{yysm}}(\text{transa}, m, n, k, \alpha, \text{descra}, \text{arg}(A), b, \text{ldb}, \beta, c, \text{ldc}, \text{work}, lwork), \text{for matrix-matrix product};
\]

\[
x_{\text{yyysm}}(\text{transa}, m, n, \text{unitd}, dv, \alpha, \text{descra}, \text{arg}(A), b, \text{ldb}, \beta, c, \text{ldc}, \text{work}, lwork), \text{for triangular solvers with multiple right-hand sides}.
\]

Some similar arguments are used in both libraries. The argument \( \text{transa} \) indicates what operation is performed and is slightly different in the NSB library (see Table "Parameter transa"). The arguments \( m \) and \( k \) are the number of rows and column in the matrix \( A \), respectively. \( n \) is the number of columns in the matrix \( C \). The arguments \( \alpha \) and \( \beta \) are scalar \( \alpha \) and \( \beta \) respectively (\( \beta \) not used in the Intel® MKL triangular solvers.) The arguments \( b \) and \( c \) are rectangular arrays with the leading dimension \( \text{ldb} \) and \( \text{ldc} \), respectively. \( \text{arg}(A) \) denotes the list of arguments that describe the sparse representation of \( A \).
**Parameter transa**

<table>
<thead>
<tr>
<th>MKL interface</th>
<th>NSB interface</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>data type</td>
<td>char *</td>
<td>INTEGER</td>
</tr>
<tr>
<td>value</td>
<td>N or n</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>T or t</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>C or c</td>
<td>2</td>
</tr>
</tbody>
</table>

**Parameter matdescra**

The parameter `matdescra` describes the relevant characteristic of the matrix \( A \). This manual describes `matdescra` as an array of six elements in line with the NIST* implementation. However, only the first four elements of the array are used in the current versions of the Intel® MKL Sparse BLAS routines. Elements `matdescra[4]` and `matdescra[5]` are reserved for future use. Note that whether `matdescra` is described in your application as an array of length 6 or 4 is of no importance because the array is declared as a pointer in the Intel® MKL routines. To learn more about declaration of the `matdescra` array, see the Sparse BLAS examples located in the Intel® MKL installation directory: examples/spblasc/ for C. The table below lists elements of the parameter `matdescra`, their Fortran values, and their meanings. The parameter `matdescra` corresponds to the argument `descra` from NSB library.

### Possible Values of the Parameter `matdescra [desca - 1]`

<table>
<thead>
<tr>
<th>MKL interface</th>
<th>NSB interface</th>
<th>Matrix characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-based</td>
<td>zero-based</td>
<td></td>
</tr>
<tr>
<td>data type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>char *</td>
<td>char *</td>
<td>int *</td>
</tr>
<tr>
<td>1st element</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>matdescra[1]</code></td>
<td><code>matdescra[0]</code></td>
<td><code>descra[0]</code></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>G</td>
<td>0</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>2nd element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>1</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>2</td>
</tr>
<tr>
<td>3rd element</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>0</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>1</td>
</tr>
</tbody>
</table>
MKL interface | NSB interface | Matrix characteristics
--- | --- | ---
value | F | 1 | one-based indexing
 | C | 0 | zero-based indexing

In some cases possible element values of the parameter matdescra depend on the values of other elements. The Table "Possible Combinations of Element Values of the Parameter matdescra" lists all possible combinations of element values for both multiplication routines and triangular solvers.

**Possible Combinations of Element Values of the Parameter matdescra**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication Routines</td>
<td>G</td>
<td>ignored</td>
<td>ignored</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>S or H</td>
<td>L (default)</td>
<td>N (default)</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>S or H</td>
<td>L (default)</td>
<td>U</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>S or H</td>
<td>U</td>
<td>N (default)</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>S or H</td>
<td>U</td>
<td>U</td>
<td>F (default) or C</td>
</tr>
<tr>
<td>A</td>
<td>L (default)</td>
<td>ignored</td>
<td>F (default) or C</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>U</td>
<td>ignored</td>
<td>F (default) or C</td>
<td></td>
</tr>
<tr>
<td>Multiplication Routines and Triangular Solvers</td>
<td>T</td>
<td>L</td>
<td>U</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>L</td>
<td>N</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>U</td>
<td>U</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>T</td>
<td>U</td>
<td>N</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>ignored</td>
<td>N (default)</td>
<td>F (default) or C</td>
</tr>
<tr>
<td></td>
<td>D</td>
<td>ignored</td>
<td>U</td>
<td>F (default) or C</td>
</tr>
</tbody>
</table>

For a matrix in the skyline format with the main diagonal declared to be a unit, diagonal elements must be stored in the sparse representation even if they are zero. In all other formats, diagonal elements can be stored (if needed) in the sparse representation if they are not zero.

**Operations with Partial Matrices**

One of the distinctive feature of the Intel® MKL Sparse BLAS routines is a possibility to perform operations only on partial matrices composed of certain parts (triangles and the main diagonal) of the input sparse matrix. It can be done by setting properly first three elements of the parameter matdescra.

An arbitrary sparse matrix \( A \) can be decomposed as

\[
A = L + D + U
\]

where \( L \) is the strict lower triangle of \( A \), \( U \) is the strict upper triangle of \( A \), \( D \) is the main diagonal.

Table "Output Matrices for Multiplication Routines" shows correspondence between the output matrices and values of the parameter matdescra for the sparse matrix \( A \) for multiplication routines.

**Output Matrices for Multiplication Routines**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>ignored</td>
<td>ignored</td>
<td>( \alpha \text{op}(A) \times x + \beta y )</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>-------------</td>
<td>---------------</td>
</tr>
<tr>
<td>S or H</td>
<td>L</td>
<td>N</td>
<td>alpha*op(A)<em>B + beta</em>C</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L+D+L')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L+D+L')<em>B + beta</em>C</td>
</tr>
<tr>
<td>S or H</td>
<td>L</td>
<td>U</td>
<td>alpha*op(L+I+L')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L+I+L')<em>B + beta</em>C</td>
</tr>
<tr>
<td>S or H</td>
<td>U</td>
<td>N</td>
<td>alpha*op(U'+D+U')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(U'+D+U')<em>B + beta</em>C</td>
</tr>
<tr>
<td>S or H</td>
<td>U</td>
<td>U</td>
<td>alpha*op(U'+I+U')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(U'+I+U')<em>B + beta</em>C</td>
</tr>
<tr>
<td>T</td>
<td>L</td>
<td>U</td>
<td>alpha*op(L+I)<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L+I)<em>B + beta</em>C</td>
</tr>
<tr>
<td>T</td>
<td>L</td>
<td>N</td>
<td>alpha*op(L+D)<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L+D)<em>B + beta</em>C</td>
</tr>
<tr>
<td>T</td>
<td>U</td>
<td>U</td>
<td>alpha*op(U+I)<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(U+I)<em>B + beta</em>C</td>
</tr>
<tr>
<td>T</td>
<td>U</td>
<td>N</td>
<td>alpha*op(U+D)<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(U+D)<em>B + beta</em>C</td>
</tr>
<tr>
<td>A</td>
<td>L</td>
<td>ignored</td>
<td>alpha*op(L-L')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(L-L')<em>B + beta</em>C</td>
</tr>
<tr>
<td>A</td>
<td>U</td>
<td>ignored</td>
<td>alpha*op(U-U')<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*op(U-U')<em>B + beta</em>C</td>
</tr>
<tr>
<td>D</td>
<td>ignored</td>
<td>N</td>
<td>alpha<em>D</em>x + beta*y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha<em>D</em>B + beta*C</td>
</tr>
<tr>
<td>D</td>
<td>ignored</td>
<td>U</td>
<td>alpha<em>x + beta</em>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha<em>B + beta</em>C</td>
</tr>
</tbody>
</table>

Table "Output Matrices for Triangular Solvers" shows correspondence between the output matrices and values of the parameter `matdescra` for the sparse matrix `A` for triangular solvers.

### Output Matrices for Triangular Solvers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>L</td>
<td>N</td>
<td>alpha*inv(op(L))*x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*inv(op(L))*B</td>
</tr>
<tr>
<td>T</td>
<td>L</td>
<td>U</td>
<td>alpha*inv(op(L))*x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*inv(op(L))*B</td>
</tr>
<tr>
<td>T</td>
<td>U</td>
<td>N</td>
<td>alpha*inv(op(U))*x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>alpha*inv(op(U))*B</td>
</tr>
</tbody>
</table>
Sparse BLAS Level 2 and Level 3 Routines.

NOTE The Intel® MKL Sparse BLAS Level 2 and Level 3 routines are deprecated. Use the corresponding routine from the Intel® MKL Inspector-executor Sparse BLAS interface as indicated in the description for each routine.

Table “Sparse BLAS Level 2 and Level 3 Routines” lists the sparse BLAS Level 2 and Level 3 routines described in more detail later in this section.

<table>
<thead>
<tr>
<th>Routine/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_?csrgemv</td>
<td>Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation)</td>
</tr>
<tr>
<td>mkl_?bsrgemv</td>
<td>Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation).</td>
</tr>
<tr>
<td>mkl_?coogemv</td>
<td>Computes matrix - vector product of a sparse general matrix in the coordinate format.</td>
</tr>
<tr>
<td>mkl_?diagemv</td>
<td>Computes matrix - vector product of a sparse general matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?csrsymv</td>
<td>Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation)</td>
</tr>
<tr>
<td>mkl_?bsrsymv</td>
<td>Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation).</td>
</tr>
<tr>
<td>mkl_?coosymv</td>
<td>Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format.</td>
</tr>
<tr>
<td>mkl_?diasymv</td>
<td>Computes matrix - vector product of a sparse symmetrical matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?csrtrsv</td>
<td>Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation).</td>
</tr>
<tr>
<td>mkl_?bsrtrsv</td>
<td>Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation).</td>
</tr>
<tr>
<td>mkl_?cootrsv</td>
<td>Triangular solvers with simplified interface for a sparse matrix in the coordinate format.</td>
</tr>
<tr>
<td>Routine/Function</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mkl_?diatrsv</td>
<td>Triangular solvers with simplified interface for a sparse matrix in the diagonal format.</td>
</tr>
</tbody>
</table>

**Simplified interface, zero-based indexing**

- **mkl_cspblas_?csrgemv**
  Computes matrix - vector product of a sparse general matrix in the CSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?bsrgemv**
  Computes matrix - vector product of a sparse general matrix in the BSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?coogemv**
  Computes matrix - vector product of a sparse general matrix in the coordinate format with zero-based indexing.

- **mkl_cspblas_?csrsymv**
  Computes matrix - vector product of a sparse symmetrical matrix in the CSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?bsrsymv**
  Computes matrix - vector product of a sparse symmetrical matrix in the BSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?coosymv**
  Computes matrix - vector product of a sparse symmetrical matrix in the coordinate format with zero-based indexing.

- **mkl_cspblas_?csrtrsv**
  Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?bsrtrsv**
  Triangular solver with simplified interface for a sparse matrix in the BSR format (3-array variation) with zero-based indexing.

- **mkl_cspblas_?cootrsv**
  Triangular solver with simplified interface for a sparse matrix in the coordinate format with zero-based indexing.

**Typical (conventional) interface, one-based and zero-based indexing**

- **mkl_?csrmv**
  Computes matrix - vector product of a sparse matrix in the CSR format.

- **mkl_?bsrmv**
  Computes matrix - vector product of a sparse matrix in the BSR format.

- **mkl_?cscmv**
  Computes matrix - vector product of a sparse matrix in the CSC format.

- **mkl_?coomv**
  Computes matrix - vector product of a sparse matrix in the coordinate format.

- **mkl_?csrsrv**
  Solves a system of linear equations for a sparse matrix in the CSR format.

- **mkl_?bsrsrv**
  Solves a system of linear equations for a sparse matrix in the BSR format.
<table>
<thead>
<tr>
<th>Routine/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_?cscsv</td>
<td>Solves a system of linear equations for a sparse matrix in the CSC format.</td>
</tr>
<tr>
<td>mkl_?coosv</td>
<td>Solves a system of linear equations for a sparse matrix in the coordinate format.</td>
</tr>
<tr>
<td>mkl_?csrmm</td>
<td>Computes matrix - matrix product of a sparse matrix in the CSR format.</td>
</tr>
<tr>
<td>mkl_?bsrmm</td>
<td>Computes matrix - matrix product of a sparse matrix in the BSR format.</td>
</tr>
<tr>
<td>mkl_?cscmm</td>
<td>Computes matrix - matrix product of a sparse matrix in the CSC format.</td>
</tr>
<tr>
<td>mkl_?coomm</td>
<td>Computes matrix - matrix product of a sparse matrix in the coordinate format.</td>
</tr>
<tr>
<td>mkl_?csrsm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the CSR format.</td>
</tr>
<tr>
<td>mkl_?bsrsm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the BSR format.</td>
</tr>
<tr>
<td>mkl_?cscsm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the CSC format.</td>
</tr>
<tr>
<td>mkl_?coosm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the coordinate format.</td>
</tr>
</tbody>
</table>

**Typical (conventional) interface, one-based indexing**

<table>
<thead>
<tr>
<th>Routine/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_?diamv</td>
<td>Computes matrix - vector product of a sparse matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?skymv</td>
<td>Computes matrix - vector product for a sparse matrix in the skyline storage format.</td>
</tr>
<tr>
<td>mkl_?diasv</td>
<td>Solves a system of linear equations for a sparse matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?skysv</td>
<td>Solves a system of linear equations for a sparse matrix in the skyline format.</td>
</tr>
<tr>
<td>mkl_?diamm</td>
<td>Computes matrix - matrix product of a sparse matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?skymm</td>
<td>Computes matrix - matrix product of a sparse matrix in the skyline storage format.</td>
</tr>
<tr>
<td>mkl_?diasm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the diagonal format.</td>
</tr>
<tr>
<td>mkl_?skysm</td>
<td>Solves a system of linear matrix equations for a sparse matrix in the skyline storage format.</td>
</tr>
</tbody>
</table>

**Auxiliary routines**

**Matrix converters**

<table>
<thead>
<tr>
<th>Routine/Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_?dnscsr</td>
<td>Converts a sparse matrix in uncompressed representation to CSR format (3-array variation) and vice versa.</td>
</tr>
<tr>
<td>Routine/Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mkl_?csradd</td>
<td>Computes the sum of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.</td>
</tr>
<tr>
<td>mkl_?csrmultcsr</td>
<td>Computes the product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing.</td>
</tr>
<tr>
<td>mkl_?csrmultd</td>
<td>Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix.</td>
</tr>
</tbody>
</table>

**mkl_?csrgemv**

*Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).*

**Syntax**

```c
void mkl_scsrgemv (const char *transa, const MKL_INT *m, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
void mkl_dcsrgemv (const char *transa, const MKL_INT *m, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);
void mkl_ccsrgemv (const char *transa, const MKL_INT *m, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcsrgemv (const char *transa, const MKL_INT *m, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);
```

**Include Files**

- `mkl.h`

**Description**

This routine is deprecated. Use `mkl_sparse_?_mv` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrgemv` routine performs a matrix-vector operation defined as

\[ y := A \cdot x \]
or

\[ y := A^T x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) sparse square matrix in the CSR format (3-array variation), \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only one-based indexing of the input arrays.

### Input Parameters

- **transa**
  Specifies the operation.
  - If \( transa = 'N' \) or \('n' \), then \( y := A x \)
  - If \( transa = 'T' \) or \('t' \) or \('C' \) or \('c' \), then \( y := A^T x \)

- **m**
  Number of rows of the matrix \( A \).

- **a**
  Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to \( values \) array description in Sparse Matrix Storage Formats for more details.

- **ia**
  Array of length \( m + 1 \), containing indices of elements in the array \( a \), such that \( ia[i] - ia[0] \) is the index in the array \( a \) of the first non-zero element from the row \( i \). The value of the last element \( ia[m] - ia[0] \) is equal to the number of non-zeros. Refer to \( rowIndex \) array description in Sparse Matrix Storage Formats for more details.

- **ja**
  Array containing the column indices plus one for each non-zero element of the matrix \( A \).
  Its length is equal to the length of the array \( a \). Refer to \( columns \) array description in Sparse Matrix Storage Formats for more details.

- **x**
  Array, size is \( m \).
  On entry, the array \( x \) must contain the vector \( x \).

### Output Parameters

- **y**
  Array, size at least \( m \).
  On exit, the array \( y \) must contain the vector \( y \).

**mkl_?bsrgemv**

Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

**Syntax**

```c
void mkl_sbsrgemv (const char *transa, const MKL_INT *m, const MKL_INT *lb, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
```
void mkl_dbsrgemv (const char *transa, const MKL_INT *m, const MKL_INT *lb, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);

void mkl_cbsrgemv (const char *transa, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zbsrgemv (const char *transa, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrgemv routine performs a matrix-vector operation defined as

\[ y := A x \]

or

\[ y := A^T x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) block sparse square matrix in the BSR format (3-array variation), \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

- **transa**
  - Specifies the operation.
  - If \( transa = 'N' \) or \( 'n' \), then the matrix-vector product is computed as \( y := A x \).
  - If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then the matrix-vector product is computed as \( y := A^T x, \)

- **m**
  - Number of block rows of the matrix \( A \).

- **lb**
  - Size of the block in the matrix \( A \).

- **a**
  - Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \times lb \).
  - Refer to \textit{values} array description in BSR Format for more details.
Array of length \((m + 1)\), containing indices of block in the array \(a\), such that \(ia[i] - ia[0]\) is the index in the array \(a\) of the first non-zero element from the row \(i\). The value of the last element \(ia[m] - ia[0]\) is equal to the number of non-zero blocks. Refer to \(rowIndex\) array description in BSR Format for more details.

Array containing the column indices plus one for each non-zero block in the matrix \(A\).

Its length is equal to the number of non-zero blocks of the matrix \(A\). Refer to \(columns\) array description in BSR Format for more details.

Array, size \((m \times lb)\).

On entry, the array \(x\) must contain the vector \(x\).

Output Parameters

Array, size at least \((m \times lb)\).

On exit, the array \(y\) must contain the vector \(y\).

**mkl_?coogemv**

*Computes matrix-vector product of a sparse general matrix stored in the coordinate format with one-based indexing (deprecated).*

**Syntax**

```c
void mkl_scoogemv (const char *transa, const MKL_INT *m, const float *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const float *x, float *y);
void mkl_dcoogemv (const char *transa, const MKL_INT *m, const double *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const double *x, double *y);
void mkl_ccoogemv (const char *transa, const MKL_INT *m, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcoogemv (const char *transa, const MKL_INT *m, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *x, MKL_Complex16 *y);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?coogemv routine performs a matrix-vector operation defined as

\[
y := A \times x
\]
or

\[ y := A^T x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) sparse square matrix in the coordinate format, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only one-based indexing of the input arrays.

**Input Parameters**

- **transa**
  Specifies the operation.
  - If `transa` = 'N' or 'n', then the matrix-vector product is computed as \( y := A x \)
  - If `transa` = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as \( y := A^T x \)

- **m**
  Number of rows of the matrix \( A \).

- **val**
  Array of length \( nnz \), contains non-zero elements of the matrix \( A \) in the arbitrary order.
  Refer to `values` array description in Coordinate Format for more details.

- **rowind**
  Array of length \( nnz \), contains the row indices plus one for each non-zero element of the matrix \( A \).
  Refer to `rows` array description in Coordinate Format for more details.

- **colind**
  Array of length \( nnz \), contains the column indices plus one for each non-zero element of the matrix \( A \). Refer to `columns` array description in Coordinate Format for more details.

- **nnz**
  Specifies the number of non-zero element of the matrix \( A \).
  Refer to `nnz` description in Coordinate Format for more details.

- **x**
  Array, size is \( m \).
  One entry, the array \( x \) must contain the vector \( x \).

**Output Parameters**

- **y**
  Array, size at least \( m \).
  On exit, the array \( y \) must contain the vector \( y \).

**mkl_?diagemv**

Computes matrix - vector product of a sparse general matrix stored in the diagonal format with one-based indexing (deprecated).
Syntax

void mkl_sdiagemv (const char *transa, const MKL_INT *m, const float *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const float *x, float *y);
void mkl_ddiagemv (const char *transa, const MKL_INT *m, const double *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const double *x, double *y);
void mkl_cdiagemv (const char *transa, const MKL_INT *m, const MKL_Complex8 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zdiagemv (const char *transa, const MKL_INT *m, const MKL_Complex16 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?diagemv routine performs a matrix-vector operation defined as

\[ y := Ax \]

or

\[ y := A^T x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) sparse square matrix in the diagonal storage format, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

**transa**

Specifies the operation.

- If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := Ax \)
- If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := A^T x, \)

**m**

Number of rows of the matrix \( A \).

**val**

Two-dimensional array of size \( \text{lval} \times \text{ndiag} \), contains non-zero diagonals of the matrix \( A \). Refer to \text{values} array description in Diagonal Storage Scheme for more details.

**lval**

Leading dimension of \( \text{val} \). Refer to \text{lval} description in Diagonal Storage Scheme for more details.
idiag

Array of length ndiag, contains the distances between main diagonal and each non-zero diagonals in the matrix A.
Refer to distance array description in Diagonal Storage Scheme for more details.

ndiag

Specifies the number of non-zero diagonals of the matrix A.

x

Array, size is m.
On entry, the array x must contain the vector x.

Output Parameters

y

Array, size at least m.
On exit, the array y must contain the vector y.

mkl_?csrsymv

Computes matrix - vector product of a sparse symmetrical matrix stored in the CSR format (3-array variation) with one-based indexing (deprecated).

Syntax

void mkl_scsrsymv (const char *uplo , const MKL_INT *m , const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y);
void mkl_dcsrsymv (const char *uplo , const MKL_INT *m , const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y);
void mkl_ccsrsymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y);
void mkl_zcsrsymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y);

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrsymv routine performs a matrix-vector operation defined as

\[ y := A^T \cdot x \]

where:

\( x \) and \( y \) are vectors,

\( A \) is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation).

**NOTE**

This routine supports only one-based indexing of the input arrays.
Input Parameters

**uplo**

Specifies whether the upper or lower triangle of the matrix $A$ is used.

If $uplo = 'U$' or 'u', then the upper triangle of the matrix $A$ is used.

If $uplo = 'L'$ or 'l', then the lower triangle of the matrix $A$ is used.

**m**

Number of rows of the matrix $A$.

**a**

Array containing non-zero elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

**ia**

Array of length $m + 1$, containing indices of elements in the array $a$, such that $ia[i] - ia[0]$ is the index in the array $a$ of the first non-zero element from the row $i$. The value of the last element $ia[m] - ia[0]$ is equal to the number of non-zero elements. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

**ja**

Array containing the column indices plus one for each non-zero element of the matrix $A$.

Its length is equal to the length of the array $a$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

**x**

Array, size is $m$.

On entry, the array $x$ must contain the vector $x$.

Output Parameters

**y**

Array, size at least $m$.

On exit, the array $y$ must contain the vector $y$.

**mkl_?bsrsymv**

Computes matrix-vector product of a sparse symmetrical matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).

**Syntax**

```c
void mkl_sbsrsymv (const char *uplo, const MKL_INT *m, const MKL_INT *lb, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
void mkl_dbsrsymv (const char *uplo, const MKL_INT *m, const MKL_INT *lb, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);
void mkl_cbsrsymv (const char *uplo, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zbsrsymv (const char *uplo, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);
```
Include Files
- mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrsymv routine performs a matrix-vector operation defined as

\[ y := A \cdot x \]

where:
- \( x \) and \( y \) are vectors,
- \( A \) is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation).

**NOTE**
This routine supports only one-based indexing of the input arrays.

Input Parameters

- **uplo**
  Specifies whether the upper or low triangle of the matrix \( A \) is considered.
  - If \( \text{uplo} = 'U' \) or 'u', then the upper triangle of the matrix \( A \) is used.
  - If \( \text{uplo} = 'L' \) or 'l', then the low triangle of the matrix \( A \) is used.

- **m**
  Number of block rows of the matrix \( A \).

- **lb**
  Size of the block in the matrix \( A \).

- **a**
  Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \cdot lb \). Refer to values array description in BSR Format for more details.

- **ia**
  Array of length \((m + 1)\), containing indices of block in the array \( a \), such that \( ia[i] - ia[0] \) is the index in the array \( a \) of the first non-zero element from the row \( i \). The value of the last element \( ia[m] - ia[0] \) is equal to the number of non-zero blocks. Refer to rowIndex array description in BSR Format for more details.

- **ja**
  Array containing the column indices plus one for each non-zero block in the matrix \( A \).
  Its length is equal to the number of non-zero blocks of the matrix \( A \). Refer to columns array description in BSR Format for more details.

- **x**
  Array, size \((m \cdot lb)\).
  On entry, the array \( x \) must contain the vector \( x \).

Output Parameters

- **y**
  Array, size at least \((m \cdot lb)\).
On exit, the array $y$ must contain the vector $y$.

**mkl_?coosymv**

*Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with one-based indexing (deprecated).*

**Syntax**

```c
void mkl_scoosymv (const char *uplo , const MKL_INT *m , const float *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , float *y );
void mkl_dcoosymv (const char *uplo , const MKL_INT *m , const double *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , double *y );
void mkl_ccosymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );
void mkl_zcoosymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex16 *x , MKL_Complex16 *y );
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The **mkl_?coosymv** routine performs a matrix-vector operation defined as

$$ y := A \times x $$

where:

- $x$ and $y$ are vectors,
- $A$ is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format.

**NOTE**

This routine supports only one-based indexing of the input arrays.

**Input Parameters**

- **uplo**
  - Specifies whether the upper or low triangle of the matrix $A$ is used.
  - If $uplo = 'U' \ or \ 'u'$, then the upper triangle of the matrix $A$ is used.
  - If $uplo = 'L' \ or \ 'l'$, then the low triangle of the matrix $A$ is used.

- **m**
  - Number of rows of the matrix $A$. 
val

Array of length \(\text{nnz}\), contains non-zero elements of the matrix \(A\) in the arbitrary order.

Refer to values array description in Coordinate Format for more details.

rowind

Array of length \(\text{nnz}\), contains the row indices plus one for each non-zero element of the matrix \(A\).

Refer to rows array description in Coordinate Format for more details.

colin

Array of length \(\text{nnz}\), contains the column indices plus one for each non-zero element of the matrix \(A\). Refer to columns array description in Coordinate Format for more details.

\(\text{nnz}\)

Specifies the number of non-zero element of the matrix \(A\).

Refer to \(\text{nnz}\) description in Coordinate Format for more details.

\(x\)

Array, size is \(m\).

On entry, the array \(x\) must contain the vector \(x\).

\textbf{Output Parameters}

\(y\)

Array, size at least \(m\).

On exit, the array \(y\) must contain the vector \(y\).

\textbf{mkl\_?diasymv}

\textit{Computes matrix - vector product of a sparse symmetrical matrix stored in the diagonal format with one-based indexing (deprecated).}

\textbf{Syntax}

```c
void mkl_sdiasymv (const char *uplo , const MKL_INT *m , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *x , float *y );
void mkl_ddiasymv (const char *uplo , const MKL_INT *m , const double *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *x , double *y );
void mkl_cdiasymv (const char *uplo , const MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8 *x , MKL_Complex8 *y );
void mkl_zdiasymv (const char *uplo , const MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *x , MKL_Complex16 *y );
```

\textbf{Include Files}

- mkl.h

\textbf{Description}

This routine is deprecated. Use mkl\_sparse\_?\_mv from the Intel\textsuperscript{®} MKL Inspector-executor Sparse BLAS interface instead.
The mkl_diasymv routine performs a matrix-vector operation defined as

\[ y := A^t x \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an upper or lower triangle of the symmetrical sparse matrix.

**NOTE**
This routine supports only one-based indexing of the input arrays.

### Input Parameters

- **uplo**
  Specifies whether the upper or low triangle of the matrix \( A \) is used.
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the low triangle of the matrix \( A \) is used.

- **m**
  Number of rows of the matrix \( A \).

- **val**
  Two-dimensional array of size \( lval \) by \( ndiag \), contains non-zero diagonals of the matrix \( A \). Refer to values array description in Diagonal Storage Scheme for more details.

- **lval**
  Leading dimension of \( val \), \( lval \geq m \). Refer to \( lval \) description in Diagonal Storage Scheme for more details.

- **idiag**
  Array of length \( ndiag \), contains the distances between main diagonal and each non-zero diagonals in the matrix \( A \).
  Refer to distance array description in Diagonal Storage Scheme for more details.

- **ndiag**
  Specifies the number of non-zero diagonals of the matrix \( A \).

- **x**
  Array, size is \( m \).
  On entry, the array \( x \) must contain the vector \( x \).

### Output Parameters

- **y**
  Array, size at least \( m \).
  On exit, the array \( y \) must contain the vector \( y \).

**mkl_csrtrsv**

Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with one-based indexing (deprecated).

**Syntax**

```c
void mkl_scsrtrsv(const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
```
void mkl_dcsrtrsv (const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);

void mkl_ccsrtrsv (const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcsrtrsv (const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl ?csrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3 array variation):

\[ A \cdot y = x \]

or

\[ A^T \cdot y = x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

**uplo**

Specifies whether the upper or lower triangle of the matrix \( A \) is used.

- If \( uplo = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.
- If \( uplo = 'L' \) or \( 'l' \), then the lower triangle of the matrix \( A \) is used.

**transa**

Specifies the system of linear equations.

- If \( transa = 'N' \) or \( 'n' \), then \( A \cdot y = x \)
- If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( A^T \cdot y = x \)

**diag**

Specifies whether \( A \) is unit triangular.

- If \( diag = 'U' \) or \( 'u' \), then \( A \) is a unit triangular.
- If \( diag = 'N' \) or \( 'n' \), then \( A \) is not unit triangular.

**m**

Number of rows of the matrix \( A \).
Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to \textit{values} array description in \textit{Sparse Matrix Storage Formats} for more details.

\textbf{NOTE}

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

\begin{itemize}
\item \textit{ia} array of length \(m + 1\), containing indices of elements in the array \(a\), such that \(ia[i] - ia[0]\) is the index in the array \(a\) of the first non-zero element from the row \(i\). The value of the last element \(ia[m] - ia[0]\) is equal to the number of non-zeros. Refer to \textit{rowIndex} array description in \textit{Sparse Matrix Storage Formats} for more details.
\item \textit{ja} array containing the column indices plus one for each non-zero element of the matrix \(A\).
\end{itemize}

\begin{itemize}
\item Its length is equal to the length of the array \(a\). Refer to \textit{columns} array description in \textit{Sparse Matrix Storage Formats} for more details.
\end{itemize}

\textbf{NOTE}

Column indices must be sorted in increasing order for each row.

\begin{itemize}
\item \textit{x} array, size is \(m\).
\end{itemize}

On entry, the array \(x\) must contain the vector \(x\).

\textbf{Output Parameters}

\begin{itemize}
\item \textit{y} array, size at least \(m\).
\end{itemize}

Contains the vector \(y\).

\textbf{mkl?bsrtrsv}

\textit{Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with one-based indexing (deprecated).}

\textbf{Syntax}

\begin{verbatim}
void mkl_sbsrtrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_INT *lb , const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_dbsrtrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_INT *lb , const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );
void mkl_cbsrtrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_INT *lb , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );
\end{verbatim}
void mkl_zbsrtrs (const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_zbsrtrs routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation):

\[ y := A \times x \]
or
\[ y := A^T \times x, \]
where:

- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only one-based indexing of the input arrays.

Input Parameters

**uplo**
Specifies the upper or lower triangle of the matrix \( A \) is used.
- If \( uplo = 'U' \) or 'u', then the upper triangle of the matrix \( A \) is used.
- If \( uplo = 'L' \) or 'l', then the lower triangle of the matrix \( A \) is used.

**transa**
Specifies the operation.
- If \( transa = 'N' \) or 'n', then the matrix-vector product is computed as \( y := A \times x \)
- If \( transa = 'T' \) or 't' or 'C' or 'c', then the matrix-vector product is computed as \( y := A^T \times x \).

**diag**
Specifies whether \( A \) is a unit triangular matrix.
- If \( diag = 'U' \) or 'u', then \( A \) is a unit triangular.
- If \( diag = 'N' \) or 'n', then \( A \) is not a unit triangular.

**m**
Number of block rows of the matrix \( A \).

**lb**
Size of the block in the matrix \( A \).

**a**
Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \times lb \). Refer to values array description in BSR Format for more details.
NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).
No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

\[ ia \]
Array of length \( (m + 1) \), containing indices of block in the array \( a \), such that \( ia[I] - ia[0] \) is the index in the array \( a \) of the first non-zero element from the row \( I \). The value of the last element \( ia[m] - ia[0] \) is equal to the number of non-zero blocks. Refer to \textit{rowIndex} array description in BSR Format for more details.

\[ ja \]
Array containing the column indices plus one for each non-zero block in the matrix \( A \).
Its length is equal to the number of non-zero blocks of the matrix \( A \). Refer to \textit{columns} array description in BSR Format for more details.

\[ x \]
Array, size \((m \times lb)\).
On entry, the array \( x \) must contain the vector \( x \).

\textbf{Output Parameters}

\[ y \]
Array, size at least \((m \times lb)\).
On exit, the array \( y \) must contain the vector \( y \).

\textbf{mkl\_cootrsv}
Triangular solvers with simplified interface for a sparse matrix in the coordinate format with one-based indexing (deprecated).

\textbf{Syntax}

```c
void mkl_scootrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const float *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , float *y );

void mkl_dcootrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const double *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , double *y );

void mkl_ccootrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );

void mkl_zcootrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex16 *x , MKL_Complex16 *y );
```

\textbf{Include Files}

- mkl.h
Description

This routine is deprecated. Use \texttt{mkl_sparse_?_trsv} from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The \texttt{mkl_?cootrsv} routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format:

\[ A^T y = x \]

or

\[ A^T y = x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

\texttt{uplo}

Specifies whether the upper or low triangle of the matrix \( A \) is considered.

- If \( \texttt{uplo} = 'U' \) or \'u', then the upper triangle of the matrix \( A \) is used.
- If \( \texttt{uplo} = 'L' \) or \'l', then the low triangle of the matrix \( A \) is used.

\texttt{transa}

Specifies the system of linear equations.

- If \( \texttt{transa} = 'N' \) or \'n', then \( A^T y = x \).
- If \( \texttt{transa} = 'T' \) or \'t' or \'C' or \'c', then \( A^T y = x \).

\texttt{diag}

Specifies whether \( A \) is unit triangular.

- If \( \texttt{diag} = 'U' \) or \'u', then \( A \) is unit triangular.
- If \( \texttt{diag} = 'N' \) or \'n', then \( A \) is not unit triangular.

\texttt{m}

Number of rows of the matrix \( A \).

\texttt{val}

Array of length \( \texttt{nnz} \), contains non-zero elements of the matrix \( A \) in the arbitrary order.

Refer to values array description in Coordinate Format for more details.

\texttt{rowind}

Array of length \( \texttt{nnz} \), contains the row indices plus one for each non-zero element of the matrix \( A \).

Refer to rows array description in Coordinate Format for more details.

\texttt{colind}

Array of length \( \texttt{nnz} \), contains the column indices plus one for each non-zero element of the matrix \( A \). Refer to columns array description in Coordinate Format for more details.

\texttt{nnz}

Specifies the number of non-zero element of the matrix \( A \).

Refer to \( \texttt{nnz} \) description in Coordinate Format for more details.
Array, size is \( m \).
On entry, the array \( x \) must contain the vector \( x \).

**Output Parameters**

\( y \)
Array, size at least \( m \).
Contains the vector \( y \).

**mkl_?diatrsv**

Triangular solvers with simplified interface for a sparse matrix in the diagonal format with one-based indexing (deprecated).

**Syntax**

```c
void mkl_sdiatrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *x , float *y );
void mkl_ddiatrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const double *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *x , double *y );
void mkl_cdiatrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8 *x , MKL_Complex8 *y );
void mkl_zdiatrsv (const char *uplo , const char *transa , const char *diag , const MKL_INT *m , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *x , MKL_Complex16 *y );
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The **mkl_?diatrsv** routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

\[
A \cdot y = x \\
\text{or} \\
A^T \cdot y = x,
\]

where:

\( x \) and \( y \) are vectors,

\( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.
Input Parameters

\underline{uplo} Specifies whether the upper or low triangle of the matrix \( A \) is used.

- If \( \text{uplo} = 'U' \) or \'u'\), then the upper triangle of the matrix \( A \) is used.
- If \( \text{uplo} = 'L' \) or \'l'\), then the low triangle of the matrix \( A \) is used.

\underline{transa} Specifies the system of linear equations.

- If \( \text{transa} = 'N' \) or \'n'\), then \( A^T \cdot y = x \)
- If \( \text{transa} = 'T' \) or \'t'\), or \'C'\) or \'c'\), then \( A^T \cdot y = x \),

\underline{diag} Specifies whether \( A \) is unit triangular.

- If \( \text{diag} = 'U' \) or \'u'\), then \( A \) is unit triangular.
- If \( \text{diag} = 'N' \) or \'n'\), then \( A \) is not unit triangular.

\underline{m} Number of rows of the matrix \( A \).

\underline{val} Two-dimensional array of size \( \text{lval} \) by \( \text{ndiag} \), contains non-zero diagonals of the matrix \( A \). Refer to \text{values} array description in \text{Diagonal Storage Scheme} for more details.

\underline{lval} Leading dimension of \( \text{val} \), \( \text{lval} \geq m \). Refer to \text{lval} description in \text{Diagonal Storage Scheme} for more details.

\underline{idiat} Array of length \( \text{ndiag} \), contains the distances between main diagonal and each non-zero diagonals in the matrix \( A \).

\text{NOTE}
All elements of this array must be sorted in increasing order.

Refer to \text{distance} array description in \text{Diagonal Storage Scheme} for more details.

\underline{ndiag} Specifies the number of non-zero diagonals of the matrix \( A \).

\underline{x} Array, size is \( m \).

On entry, the array \( x \) must contain the vector \( x \).

Output Parameters

\underline{y} Array, size at least \( m \).

Contains the vector \( y \).

\text{mkl_cspblas_?csrgemv}
Computes matrix - vector product of a sparse general matrix stored in the CSR format (3-array variation) with zero-based indexing (deprecated).

\text{Syntax}

\begin{verbatim}
void mkl_cspblas_scsrgemv (const char *transa, const MKL_INT *m, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
\end{verbatim}
void mkl_cspblas_dcsrgemv (const char *transa, const MKL_INT *m, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);
void mkl_cspblas_ccsrgemv (const char *transa, const MKL_INT *m, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zcsrgemv (const char *transa, const MKL_INT *m, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrgemv routine performs a matrix-vector operation defined as

\[ y := A \cdot x \]

or

\[ y := A^T \cdot x, \]

where:
- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) sparse square matrix in the CSR format (3-array variation) with zero-based indexing, \( A^T \) is the transpose of \( A \).

NOTE
This routine supports only zero-based indexing of the input arrays.

Input Parameters

transa

Specifies the operation.
- If \( \text{transa} = 'N' \) or \( 'n' \), then the matrix-vector product is computed as \( y := A \cdot x \).
- If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then the matrix-vector product is computed as \( y := A^T \cdot x \).

m

Number of rows of the matrix \( A \).

a

Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to \( \text{values} \) array description in Sparse Matrix Storage Formats for more details.

ia

Array of length \( m + 1 \), containing indices of elements in the array \( a \), such that \( ia[I] \) is the index in the array \( a \) of the first non-zero element from the row \( I \). The value of the last element \( ia[m] \) is equal to the number of non-zeros. Refer to \( \text{rowIndex} \) array description in Sparse Matrix Storage Formats for more details.
Array containing the column indices for each non-zero element of the matrix A.
Its length is equal to the length of the array a. Refer to columns array description in Sparse Matrix Storage Formats for more details.

Array, size is m.
One entry, the array x must contain the vector x.

Output Parameters

Array, size at least m.
On exit, the array y must contain the vector y.

**mkl_cspblas_?bsrgemv**

*Computes matrix - vector product of a sparse general matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).*

**Syntax**

```c
void mkl_cspblas_sbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_cspblas_dbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );
void mkl_cspblas_cbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );
void mkl_cspblas_zbsrgemv (const char *transa , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *
*x , MKL_Complex16 *y );
```

**Include Files**
- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?bsrgemv routine performs a matrix-vector operation defined as

\[ y := A \times x \]

or

\[ y := A^T \times x , \]

where:

x and y are vectors,

A is an m-by-m block sparse square matrix in the BSR format (3-array variation) with zero-based indexing, 
\( A^T \) is the transpose of A.
NOTE
This routine supports only zero-based indexing of the input arrays.

**Input Parameters**

*transa*  
Specifies the operation.
If *transa* = 'N' or 'n', then the matrix-vector product is computed as  
y := A*x
If *transa* = 'T' or 't' or 'C' or 'c', then the matrix-vector product is computed as  
y := A^T*x,

*m*  
Number of block rows of the matrix A.

*lb*  
Size of the block in the matrix A.

*a*  
Array containing elements of non-zero blocks of the matrix A. Its length is equal to the number of non-zero blocks in the matrix A multiplied by *lb*.*lb*. Refer to values array description in BSR Format for more details.

*ia*  
Array of length (*m* + 1), containing indices of block in the array a, such that *ia[i]* is the index in the array a of the first non-zero element from the row i. The value of the last element *ia[m]* is equal to the number of non-zero blocks. Refer to rowIndex array description in BSR Format for more details.

*ja*  
Array containing the column indices for each non-zero block in the matrix A. Its length is equal to the number of non-zero blocks of the matrix A. Refer to columns array description in BSR Format for more details.

*x*  
Array, size (*m*.*lb*).
On entry, the array *x* must contain the vector *x*.

**Output Parameters**

*y*  
Array, size at least (*m*.*lb*).
On exit, the array *y* must contain the vector *y*.

**mkl_cspblas_?coogemv**

Computes matrix - vector product of a sparse general matrix stored in the coordinate format with zero-based indexing (deprecated).

**Syntax**

```c
void mkl_cspblas_scoogemv (const char *transa , const MKL_INT *m , const float *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x ,
float *y );
void mkl_cspblas_dcoogemv (const char *transa , const MKL_INT *m , const double *val ,
const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x ,
double *y );
```
void mkl_cspblas_ccoogemv (const char *transa, const MKL_INT *m, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcoogemv (const char *transa, const MKL_INT *m, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_dcoogemv routine performs a matrix-vector operation defined as

\[ y := Ax \]

or

\[ y := A^T x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( m \) sparse square matrix in the coordinate format with zero-based indexing, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only zero-based indexing of the input arrays.

Input Parameters

**transa**

Specifies the operation.

- If \( \text{transa} = 'N' \) or \( 'n' \), then the matrix-vector product is computed as \( y := Ax \).
- If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then the matrix-vector product is computed as \( y := A^T x \).

**m**

Number of rows of the matrix \( A \).

**val**

Array of length \( \text{nnz} \), contains non-zero elements of the matrix \( A \) in the arbitrary order.

Refer to **values** array description in Coordinate Format for more details.

**rowind**

Array of length \( \text{nnz} \), contains the row indices for each non-zero element of the matrix \( A \).

Refer to **rows** array description in Coordinate Format for more details.

**colind**

Array of length \( \text{nnz} \), contains the column indices for each non-zero element of the matrix \( A \). Refer to **columns** array description in Coordinate Format for more details.
Specifies the number of non-zero element of the matrix $A$.
Refer to $nnz$ description in Coordinate Format for more details.

$x$
Array, size is $m$.
On entry, the array $x$ must contain the vector $x$.

Output Parameters

$y$
Array, size at least $m$.
On exit, the array $y$ must contain the vector $y$.

**mkl_cspblas_?csrsymv**

Computes matrix-vector product of a sparse
symmetrical matrix stored in the CSR format (3-array variation) with zero-based indexing (deprecated).

**Syntax**

void mkl_cspblas_scsrsymv (const char *uplo, const MKL_INT *m, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *x, float *y);
void mkl_cspblas_dcsrsymv (const char *uplo, const MKL_INT *m, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *x, double *y);
void mkl_cspblas_ccsrsymv (const char *uplo, const MKL_INT *m, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zcsrsymv (const char *uplo, const MKL_INT *m, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *x, MKL_Complex16 *y);

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrsymv routine performs a matrix-vector operation defined as

$$y := A \times x$$

where:

$x$ and $y$ are vectors,

$A$ is an upper or lower triangle of the symmetrical sparse matrix in the CSR format (3-array variation) with zero-based indexing.

**NOTE**

This routine supports only zero-based indexing of the input arrays.

**Input Parameters**
uplo

Specifies whether the upper or low triangle of the matrix A is used.
If uplo = 'U' or 'u', then the upper triangle of the matrix A is used.
If uplo = 'L' or 'l', then the low triangle of the matrix A is used.

m

Number of rows of the matrix A.

a

Array containing non-zero elements of the matrix A. Its length is equal to
the number of non-zero elements in the matrix A. Refer to values array
description in Sparse Matrix Storage Formats for more details.

ia

Array of length m + 1, containing indices of elements in the array a, such
that ia[i] is the index in the array a of the first non-zero element from the
row i. The value of the last element ia[m] is equal to the number of non-
zeros. Refer to rowIndex array description in Sparse Matrix Storage
Formats for more details.

ja

Array containing the column indices for each non-zero element of the
matrix A.
Its length is equal to the length of the array a. Refer to columns array
description in Sparse Matrix Storage Formats for more details.

x

Array, size is m.
On entry, the array x must contain the vector x.

Output Parameters

y

Array, size at least m.
On exit, the array y must contain the vector y.

mkl_cspblas_?bsrsymv

Computes matrix-vector product of a sparse
symmetrical matrix stored in the BSR format (3-arrays
variation) with zero-based indexing (deprecated).

Syntax

void mkl_cspblas_sbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const float *a , const MKL_INT *ia , const MKL_INT *ja , const float *x , float *y );
void mkl_cspblas_dbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const double *a , const MKL_INT *ia , const MKL_INT *ja , const double *x , double *y );
void mkl_cspblas_cbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex8 *x ,
MKL_Complex8 *y );
void mkl_cspblas_zbsrsymv (const char *uplo , const MKL_INT *m , const MKL_INT *lb ,
const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_Complex16 *
x , MKL_Complex16 *y );

Include Files

• mkl.h
Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?bsrsymv routine performs a matrix-vector operation defined as

\[ y := Ax \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is an upper or lower triangle of the symmetrical sparse matrix in the BSR format (3-array variation) with zero-based indexing.

**NOTE**
This routine supports only zero-based indexing of the input arrays.

Input Parameters

- **uplo**
  Specifies whether the upper or low triangle of the matrix \( A \) is used.
  - If \( uplo = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.
  - If \( uplo = 'L' \) or \( 'l' \), then the low triangle of the matrix \( A \) is used.

- **m**
  Number of block rows of the matrix \( A \).

- **lb**
  Size of the block in the matrix \( A \).

- **a**
  Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb*lb \). Refer to values array description in BSR Format for more details.

- **ia**
  Array of length \((m + 1)\), containing indices of block in the array \( a \), such that \( ia[i] \) is the index in the array \( a \) of the first non-zero element from the row \( i \). The value of the last element \( ia[m] \) is equal to the number of non-zero blocks. Refer to rowIndex array description in BSR Format for more details.

- **ja**
  Array containing the column indices for each non-zero block in the matrix \( A \).
  Its length is equal to the number of non-zero blocks of the matrix \( A \). Refer to columns array description in BSR Format for more details.

- **x**
  Array, size \((m*lb)\).
  On entry, the array \( x \) must contain the vector \( x \).

Output Parameters

- **y**
  Array, size at least \((m*lb)\).
  On exit, the array \( y \) must contain the vector \( y \).
mkl_cspblas_?coosymv
Computes matrix - vector product of a sparse symmetrical matrix stored in the coordinate format with zero-based indexing (deprecated).

Syntax
void mkl_cspblas_scoosymv (const char *uplo, const MKL_INT *m, const float *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const float *x, float *y);

void mkl_cspblas_dcoosymv (const char *uplo, const MKL_INT *m, const double *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const double *x, double *y);

void mkl_cspblas_ccoosymv (const char *uplo, const MKL_INT *m, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_cspblas_zcoosymv (const char *uplo, const MKL_INT *m, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *x, MKL_Complex16 *y);

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mvfrom the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?coosymv routine performs a matrix-vector operation defined as

\[ y := A \times x \]

where:

\( x \) and \( y \) are vectors,

\( A \) is an upper or lower triangle of the symmetrical sparse matrix in the coordinate format with zero-based indexing.

**NOTE**
This routine supports only zero-based indexing of the input arrays.

Input Parameters

**uplo**
Specifies whether the upper or low triangle of the matrix \( A \) is used.
If \( uplo = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.
If \( uplo = 'L' \) or \( 'l' \), then the low triangle of the matrix \( A \) is used.

**m**
Number of rows of the matrix \( A \).

**val**
Array of length \( nnz \), contains non-zero elements of the matrix \( A \) in the arbitrary order.
Refer to values array description in Coordinate Format for more details.

rowind
Array of length nnz, contains the row indices for each non-zero element of the matrix A.
Refer to rows array description in Coordinate Format for more details.

colind
Array of length nnz, contains the column indices for each non-zero element of the matrix A. Refer to columns array description in Coordinate Format for more details.

nnz
Specifies the number of non-zero element of the matrix A.
Refer to nnz description in Coordinate Format for more details.

x
Array, size is m.
On entry, the array x must contain the vector x.

Output Parameters

y
Array, size at least m.
On exit, the array y must contain the vector y.

mkl_cspblas_?csrtrsv
Triangular solvers with simplified interface for a sparse matrix in the CSR format (3-array variation) with zero-based indexing (deprecated).

Syntax

void mkl_cspblas_scsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const float *a , const MKL_INT *ia , const MKL_INT *ja ,
const float *x , float *y);
void mkl_cspblas_dcsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const double *a , const MKL_INT *ia , const MKL_INT *ja , const
double *x , double *y);
void mkl_cspblas_ccsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_Complex8 *x , MKL_Complex8 *y );
void mkl_cspblas_zcsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja ,
const MKL_Complex16 *x , MKL_Complex16 *y );

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_cspblas_?csrtrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the CSR format (3-array variation) with zero-based indexing:

\[ A^y = x \]
or

\[ A^T y = x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only zero-based indexing of the input arrays.

### Input Parameters

- **uplo**
  Specifies whether the upper or low triangle of the matrix \( A \) is used.
  - If \( uplo = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.
  - If \( uplo = 'L' \) or \( 'l' \), then the low triangle of the matrix \( A \) is used.

- **transa**
  Specifies the system of linear equations.
  - If \( transa = 'N' \) or \( 'n' \), then \( A^T y = x \)
  - If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( A^T y = x \)

- **diag**
  Specifies whether matrix \( A \) is unit triangular.
  - If \( diag = 'U' \) or \( 'u' \), then \( A \) is unit triangular.
  - If \( diag = 'N' \) or \( 'n' \), then \( A \) is not unit triangular.

- **m**
  Number of rows of the matrix \( A \).

- **a**
  Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to values array description in Sparse Matrix Storage Formats for more details.

**NOTE**
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

- **ia**
  Array of length \( m+1 \), containing indices of elements in the array \( a \), such that \( ia[i] \) is the index in the array \( a \) of the first non-zero element from the row \( i \). The value of the last element \( ia[m] \) is equal to the number of non-zeros. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

- **ja**
  Array containing the column indices for each non-zero element of the matrix \( A \).
  Its length is equal to the length of the array \( a \). Refer to columns array description in Sparse Matrix Storage Formats for more details.
Column indices must be sorted in increasing order for each row.

X

Array, size is \( m \).

On entry, the array \( x \) must contain the vector \( x \).

Output Parameters

Y

Array, size at least \( m \).

Contains the vector \( y \).

**mkl_cspblas_?bsrtrsv**

Triangular solver with simplified interface for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing (deprecated).

**Syntax**

```c
void mkl_cspblas_sbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const float *a , const MKL_INT *ia ,
const MKL_INT *ja , const float *x , float *y );
void mkl_cspblas_dbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const double *a , const MKL_INT *ia ,
const MKL_INT *ja , const double *x , double *y );
void mkl_cspblas_cbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const MKL_Complex8 *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_Complex8 *x , MKL_Complex8 *y );
void mkl_cspblas_zbsrtrsv (const char *uplo , const char *transa , const char *diag ,
const MKL_INT *m , const MKL_INT *lb , const MKL_Complex16 *a , const MKL_INT *ia ,
const MKL_INT *ja , const MKL_Complex16 *x , MKL_Complex16 *y );
```

**Include Files**

- `mkl.h`

**Description**

This routine is deprecated. Use `mkl_sparse_?_trsv` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_cspblas_?bsrtrsv` routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the BSR format (3-array variation) with zero-based indexing:

\[
y := A x
\]

or

\[
y := A^T x,
\]

where:

\( x \) and \( y \) are vectors,

\( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).
Input Parameters

**uplo**

Specifies the upper or low triangle of the matrix \( A \) is used.

If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangle of the matrix \( A \) is used.

If \( \text{uplo} = 'L' \) or \( 'l' \), then the low triangle of the matrix \( A \) is used.

**transa**

Specifies the operation.

If \( \text{transa} = 'N' \) or \( 'n' \), then the matrix-vector product is computed as \( y := A \times x \)

If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then the matrix-vector product is computed as \( y := A^T \times x \).

**diag**

Specifies whether matrix \( A \) is unit triangular or not.

If \( \text{diag} = 'U' \) or \( 'u' \), \( A \) is unit triangular.

If \( \text{diag} = 'N' \) or \( 'n' \), \( A \) is not unit triangular.

**m**

Number of block rows of the matrix \( A \).

**lb**

Size of the block in the matrix \( A \).

**a**

Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \times lb \). Refer to \textit{values} array description in BSR Format for more details.

**ia**

Array of length \( (m + 1) \), containing indices of block in the array \( a \), such that \( ia[I] \) is the index in the array \( a \) of the first non-zero element from the row \( I \). The value of the last element \( ia[m] \) is equal to the number of non-zero blocks. Refer to \textit{rowIndex} array description in BSR Format for more details.

**ja**

Array containing the column indices for each non-zero block in the matrix \( A \).

Its length is equal to the number of non-zero blocks of the matrix \( A \). Refer to \textit{columns} array description in BSR Format for more details.

**x**

Array, size \( (m \times lb) \).

On entry, the array \( x \) must contain the vector \( x \).
Output Parameters

\( y \)

Array, size at least \((m \times lb)\).

On exit, the array \( y \) must contain the vector \( y \).

\textbf{mkl_cspblas_?cootrsv}

\textit{Triangular solvers with simplified interface for a sparse matrix in the coordinate format with zero-based indexing (deprecated).}

\textbf{Syntax}

\begin{verbatim}
void mkl_cspblas_scootrsv (const char *uplo, const char *transa, const char *diag,
const MKL_INT *m, const float *val, const MKL_INT *rowind, const MKL_INT *colind,
const MKL_INT *nnz, const float *x, float *y);
void mkl_cspblas_dcootrsv (const char *uplo, const char *transa, const char *diag,
const MKL_INT *m, const double *val, const MKL_INT *rowind, const MKL_INT *colind,
const MKL_INT *nnz, const double *x, double *y);
void mkl_cspblas_ccootrsv (const char *uplo, const char *transa, const char *diag,
const MKL_INT *m, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT
*colind, const MKL_INT *nnz, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_cspblas_zcootrsv (const char *uplo, const char *transa, const char *diag,
const MKL_INT *m, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT
*colind, const MKL_INT *nnz, const MKL_Complex16 *x, MKL_Complex16 *y);
\end{verbatim}

\textbf{Include Files}

- mkl.h

\textbf{Description}

This routine is deprecated. Use \texttt{mkl_sparse_?_trsv} from the Intel\textsuperscript{®} MKL Inspector-executor Sparse BLAS interface instead.

The \texttt{mkl_cspblas_?cootrsv} routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the coordinate format with zero-based indexing:

\[ A \cdot y = x \]

or

\[ A^T \cdot y = x, \]

where:

- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

\textbf{NOTE}

This routine supports only zero-based indexing of the input arrays.

\textbf{Input Parameters}
Specifies whether the upper or low triangle of the matrix A is considered.
If uplo = 'U' or 'u', then the upper triangle of the matrix A is used.
If uplo = 'L' or 'l', then the low triangle of the matrix A is used.

**transa**
Specifies the system of linear equations.
If transa = 'N' or 'n', then $A^T y = x$
If transa = 'T' or 't' or 'C' or 'c', then $A^T y = x$

**diag**
Specifies whether A is unit triangular.
If diag = 'U' or 'u', then A is unit triangular.
If diag = 'N' or 'n', then A is not unit triangular.

**m**
Number of rows of the matrix A.

**val**
Array of length nnz, contains non-zero elements of the matrix A in the arbitrary order.
Refer to values array description in Coordinate Format for more details.

**rowind**
Array of length nnz, contains the row indices for each non-zero element of the matrix A.
Refer to rows array description in Coordinate Format for more details.

**colind**
Array of length nnz, contains the column indices for each non-zero element of the matrix A. Refer to columns array description in Coordinate Format for more details.

**nnz**
Specifies the number of non-zero element of the matrix A.
Refer to nnz description in Coordinate Format for more details.

**x**
Array, size is m.
On entry, the array x must contain the vector x.

**Output Parameters**

**y**
Array, size at least m.
Contains the vector y.

**mkl_?csrmv**
Computes matrix - vector product of a sparse matrix stored in the CSR format (deprecated).

**Syntax**

```c
void mkl_scsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *index, const MKL_INT *pntrb, const MKL_INT *pntre, const float *x, const float *beta, float *y);
void mkl_dcsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *index, const MKL_INT *pntrb, const MKL_INT *pntre, const double *x, const double *beta, double *y);
```
void mkl_ccsr
  m (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnte, const MKL_Complex8 *x, const MKL_Complex8 *beta, MKL_Complex8 *y);

void mkl_zcsr
  m (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnte, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);

Include Files
  • mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmv routine performs a matrix-vector operation defined as

\[ y := \alpha A x + \beta y \]

or

\[ y := \alpha A^T x + \beta y, \]

where:
- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( k \) sparse matrix in the CSR format, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

- **transa**
  Specifies the operation.
  - If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha A x + \beta y \)
  - If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha A^T x + \beta y \),

- **m**
  Number of rows of the matrix \( A \).

- **k**
  Number of columns of the matrix \( A \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **matdescra**
  Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**
  Array containing non-zero elements of the matrix \( A \).
  - Its length is \( \text{pnte}[m-1] - \text{pntrb}[0] \).
Refer to *values* array description in **CSR Format** for more details.

**indx**
For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix \( A \). For zero-based indexing, array containing the column indices for each non-zero element of the matrix \( A \).

Its length is equal to length of the *val* array.

Refer to *columns* array description in **CSR Format** for more details.

**pntrb**
Array of length \( m \).

This array contains row indices, such that \( pntrb[i] - pntrb[0] \) is the first index of row \( i \) in the arrays *val* and *indx*.

Refer to *pointerb* array description in **CSR Format** for more details.

**pntre**
Array of length \( m \).

This array contains row indices, such that \( pntre[i] - pntrb[0]-1 \) is the last index of row \( i \) in the arrays *val* and *indx*.

Refer to *pointerE* array description in **CSR Format** for more details.

**x**
Array, size at least \( k \) if \( transa = 'N' \) or \( 'n' \) and at least \( m \) otherwise. On entry, the array *x* must contain the vector *x*.

**beta**
Specifies the scalar *beta*.

**y**
Array, size at least \( m \) if \( transa = 'N' \) or \( 'n' \) and at least \( k \) otherwise. On entry, the array *y* must contain the vector *y*.

**Output Parameters**

*y*
Overwritten by the updated vector *y*.

**mkl_?bsrmv**
*Computes matrix - vector product of a sparse matrix stored in the BSR format (deprecated).*

**Syntax**

```c
void mkl_sbsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_INT *lb, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *x, const float *beta, float *y);

void mkl_dbsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_INT *lb, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *x, const double *beta, double *y);

void mkl_cbsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_INT *lb, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *x, const MKL_Complex8 *beta, MKL_Complex8 *y);
```
void mkl_zbsrmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_INT *lb, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrmv routine performs a matrix-vector operation defined as

\[ y := \alpha A x + \beta y \]

or

\[ y := \alpha A^T x + \beta y , \]

where:
- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( k \) block sparse matrix in the BSR format, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

- **transa**
  Specifies the operation.
  If \( transa = 'N' \) or \'n' \), then the matrix-vector product is computed as \( y := \alpha A x + \beta y \)
  If \( transa = 'T' \) or \'t' \) or \'C' \) or \'c' \), then the matrix-vector product is computed as \( y := \alpha A^T x + \beta y , \)

- **m**
  Number of block rows of the matrix \( A \).

- **k**
  Number of block columns of the matrix \( A \).

- **lb**
  Size of the block in the matrix \( A \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **matdescra**
  Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**
  Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \times lb \).
Refer to *values* array description in BSR Format for more details.

**indx**
For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix A. For zero-based indexing, array containing the column indices for each non-zero block of the matrix A.
Its length is equal to the number of non-zero blocks in the matrix A.
Refer to *columns* array description in BSR Format for more details.

**pntrb**
Array of length *m*.
This array contains row indices, such that \( pntrb[i] - pntrb[0] \) is the first index of block row \( i \) in the array *indx*
Refer to *pointerB* array description in BSR Format for more details.

**pntrc**
Array of length *m*.
For zero-based indexing this array contains row indices, such that \( pntre[i] - pntrb[0] - 1 \) is the last index of block row \( i \) in the array *indx*.
Refer to *pointerE* array description in BSR Format for more details.

**x**
Array, size at least \((k*lb)\) if transa = 'N' or 'n', and at least \((m*lb)\) otherwise. On entry, the array *x* must contain the vector *x*.

**beta**
Specifies the scalar *beta*.

**y**
Array, size at least \((m*lb)\) if transa = 'N' or 'n', and at least \((k*lb)\) otherwise. On entry, the array *y* must contain the vector *y*.

**Output Parameters**

*y*
Overwritten by the updated vector *y*.

**mkl_scscmv**
*Computes matrix-vector product for a sparse matrix in the CSC format (deprecated).*

**Syntax**

```c
void mkl_scscmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *x, const float *beta, float *y);
void mkl_dcscmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *x, const double *beta, double *y);
void mkl_ccscmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *x, const MKL_Complex8 *beta, MKL_Complex8 *y);
```
void mkl_zcscmv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?cscmv routine performs a matrix-vector operation defined as

\[ y := \alpha A x + \beta y \]

or

\[ y := \alpha A^T x + \beta y, \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( k \) sparse matrix in compressed sparse column (CSC) format, \( A^T \) is the transpose of \( A \).

NOTE

This routine supports CSC format both with one-based indexing and zero-based indexing.

Input Parameters

transa

Specifies the operation.

- If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha A x + \beta y \)
- If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha A^T x + \beta y \),

m

Number of rows of the matrix \( A \).

k

Number of columns of the matrix \( A \).

alpha

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

val

Array containing non-zero elements of the matrix \( A \).

Its length is \( \text{pntr}[k-1] - \text{pntr}[0] \).

Refer to values array description in CSC Format for more details.
For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix \(A\). For zero-based indexing, array containing the column indices for each non-zero element of the matrix \(A\).

Its length is equal to length of the \(val\) array.

Refer to \(rows\) array description in CSC Format for more details.

\(pntrb\)

*Array of length \(k\).*

This array contains column indices, such that \(pntrb[i] - pntrb[0] + 1\) is the first index of column \(i\) in the arrays \(val\) and \(indx\).

Refer to \(pointerb\) array description in CSC Format for more details.

\(pnre\)

*Array of length \(k\).*

For one-based indexing this array contains column indices, such that \(pnre[i] - pntrb[1]\) is the last index of column \(i\) in the arrays \(val\) and \(indx\).

For zero-based indexing this array contains column indices, such that \(pnre[i] - pntrb[1] - 1\) is the last index of column \(i\) in the arrays \(val\) and \(indx\).

Refer to \(pointerE\) array description in CSC Format for more details.

\(x\)

*Array, size at least \(k\) if \(transa = 'N'\) or \('n'\) and at least \(m\) otherwise. On entry, the array \(x\) must contain the vector \(x\).*

\(beta\)

*Specifies the scalar \(beta\).*

\(y\)

*Array, size at least \(m\) if \(transa = 'N'\) or \('n'\) and at least \(k\) otherwise. On entry, the array \(y\) must contain the vector \(y\).*

**Output Parameters**

\(y\)

*Overwritten by the updated vector \(y\).*

**mkl\_?coomv**

*Computes matrix - vector product for a sparse matrix in the coordinate format (deprecated).*

**Syntax**

```c
void mkl_scoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , const float *beta , float *y ) ;
void mkl_dcoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , const double *beta , double *y ) ;
void mkl_ccoomv (const char *transa , const MKL_INT *m , const MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *x , const MKL_Complex8 *beta , MKL_Complex8 *y ) ;
```
void mkl_zcoomv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);

Include Files
- mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?coomv routine performs a matrix-vector operation defined as

\[ y := \alpha A x + \beta y \]

or

\[ y := \alpha A^T x + \beta y, \]

where:
- \( \alpha \) and \( \beta \) are scalars,
- \( x \) and \( y \) are vectors,
- \( A \) is an \( m \)-by-\( k \) sparse matrix in compressed coordinate format, \( A^T \) is the transpose of \( A \).

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

\textbf{transa} 
Specifies the operation.
If \textit{transa} = 'N' or 'n', then \( y := \alpha A x + \beta y \)
If \textit{transa} = 'T' or 't' or 'C' or 'c', then \( y := \alpha A^T x + \beta y, \)

\textbf{m} 
Number of rows of the matrix \( A \).

\textbf{k} 
Number of columns of the matrix \( A \).

\textbf{alpha} 
Specifies the scalar \( \alpha \).

\textbf{matdescra} 
Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

\textbf{val} 
Array of length \( \text{nnz} \), contains non-zero elements of the matrix \( A \) in the arbitrary order. Refer to values array description in \textit{Coordinate Format} for more details.

\textbf{rowind} 
Array of length \( \text{nnz} \).
For one-based indexing, contains the row indices plus one for each non-zero element of the matrix $A$.

For zero-based indexing, contains the row indices for each non-zero element of the matrix $A$.

Refer to rows array description in Coordinate Format for more details.

colind

Array of length $nnz$.

For one-based indexing, contains the column indices plus one for each non-zero element of the matrix $A$.

For zero-based indexing, contains the column indices for each non-zero element of the matrix $A$.

Refer to columns array description in Coordinate Format for more details.

$nnz$

Specifies the number of non-zero element of the matrix $A$.

Refer to $nnz$ description in Coordinate Format for more details.

$x$

Array, size at least $k$ if $transa = 'N'$ or 'n' and at least $m$ otherwise. On entry, the array $x$ must contain the vector $x$.

beta

Specifies the scalar $beta$.

$y$

Array, size at least $m$ if $transa = 'N'$ or 'n' and at least $k$ otherwise. On entry, the array $y$ must contain the vector $y$.

Output Parameters

$y$

Overwritten by the updated vector $y$.

**mkl?csrsv**

Solves a system of linear equations for a sparse matrix in the CSR format (deprecated).

**Syntax**

```c
void mkl_scsrsv (const char *transa, const MKL_INT *m, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *x, float *y);
void mkl_dcsrsv (const char *transa, const MKL_INT *m, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *x, double *y);
void mkl_ccsrsv (const char *transa, const MKL_INT *m, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zcsrsv (const char *transa, const MKL_INT *m, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *x, MKL_Complex16 *y);
```

**Include Files**

- mkl.h
Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSR format:

\[ y := \alpha \cdot \text{inv}(A) \cdot x \]

or

\[ y := \alpha \cdot \text{inv}(A^T) \cdot x, \]

where:

\( \alpha \) is scalar, \( x \) and \( y \) are vectors, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

**transa**

Specifies the system of linear equations.

If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha \cdot \text{inv}(A) \cdot x \)

If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha \cdot \text{inv}(A^T) \cdot x \),

**m**

Number of columns of the matrix \( A \).

**alpha**

Specifies the scalar \( \alpha \).

**matdescra**

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**val**

Array containing non-zero elements of the matrix \( A \).

Its length is \( \text{pntr}[m - 1] - \text{pntr}[0] \).

Refer to \textit{values} array description in CSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

**indx**

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix \( A \). For zero-based indexing, array containing the column indices for each non-zero element of the matrix \( A \).

Its length is equal to length of the \textit{val} array.
Refer to *columns* array description in **CSR Format** for more details.

**NOTE**
Column indices must be sorted in increasing order for each row.

$pntrb$

Array of length $m$.
This array contains row indices, such that $pntrb[i] - pntrb[0]$ is the first index of row $i$ in the arrays $val$ and $indx$.
Refer to *pointerb* array description in **CSR Format** for more details.

$pnTre$

Array of length $m$.
This array contains row indices, such that $pnTre[i] - pntrb[0] - 1$ is the last index of row $i$ in the arrays $val$ and $indx$.
Refer to *pointerE* array description in **CSR Format** for more details.

$x$

Array, size at least $m$.
On entry, the array $x$ must contain the vector $x$. The elements are accessed with unit increment.

$y$

Array, size at least $m$.
On entry, the array $y$ must contain the vector $y$. The elements are accessed with unit increment.

**Output Parameters**

$y$

Contains solution vector $x$.

### mkl_?bsrsv

*Solves a system of linear equations for a sparse matrix in the BSR format (deprecated).*

**Syntax**

```c
void mkl_sbsrsv(const char *transa, const MKL_INT *m, const MKL_INT *lb, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnTre, const float *x, float *y);
void mkl_dbsrsv(const char *transa, const MKL_INT *m, const MKL_INT *lb, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnTre, const double *x, double *y);
void mkl_cbsrsv(const char *transa, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnTre, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zbsrsv(const char *transa, const MKL_INT *m, const MKL_INT *lb, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pnTre, const MKL_Complex16 *x, MKL_Complex16 *y);
```
Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the BSR format:

\[ y := \alpha \cdot \text{inv}(A) \cdot x \]

or

\[ y := \alpha \cdot \text{inv}(A^T) \cdot x, \]

where:

- \( \alpha \) is scalar,
- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal,
- \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

- **transa**: Specifies the operation.
  - If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha \cdot \text{inv}(A) \cdot x \)
  - If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha \cdot \text{inv}(A^T) \cdot x \)

- **m**: Number of block columns of the matrix \( A \).

- **lb**: Size of the block in the matrix \( A \).

- **alpha**: Specifies the scalar \( \alpha \).

- **matdescra**:
  - Array of six elements, specifies properties of the matrix used for operation.
  - Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**: Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( lb \cdot lb \).

**NOTE**
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.
For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix $A$. For zero-based indexing, array containing the column indices for each non-zero element of the matrix $A$. Its length is equal to the number of non-zero blocks in the matrix $A$. Refer to the columns array description in BSR Format for more details.

Array of length $m$.

This array contains row indices, such that $pntrb[i] - pntrb[0]$ is the first index of block row $i$ in the array $indx$.

Refer to pointerB array description in BSR Format for more details.

Array of length $m$.

For one-based indexing this array contains row indices, such that $pntre[i] - pntrb[1]$ is the last index of block row $i$ in the array $indx$.

For zero-based indexing this array contains row indices, such that $pntre[i] - pntrb[0] - 1$ is the last index of block row $i$ in the array $indx$.

Refer to pointerE array description in BSR Format for more details.

Array, size at least $(m*lb)$.

On entry, the array x must contain the vector $x$. The elements are accessed with unit increment.

Array, size at least $(m*lb)$.

On entry, the array $y$ must contain the vector $y$. The elements are accessed with unit increment.

Output Parameters

$y$

Contains solution vector $x$.

mkl_?cscsv

Solves a system of linear equations for a sparse matrix in the CSC format (deprecated).

Syntax

```c
void mkl_scscsv (const char *transa, const MKL_INT *m, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *x, float *y);

void mkl_dcscsv (const char *transa, const MKL_INT *m, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *x, double *y);

void mkl_ccscsv (const char *transa, const MKL_INT *m, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *x, MKL_Complex8 *y);

void mkl_zcscsv (const char *transa, const MKL_INT *m, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *x, MKL_Complex16 *y);
```
Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?cscsv routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the CSC format:

\[ y := \alpha \cdot \text{inv}(A) \cdot x \]

or

\[ y := \alpha \cdot \text{inv}(A^T) \cdot x, \]

where:

- \( \alpha \) is scalar,
- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal,
- \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports a CSC format both with one-based indexing and zero-based indexing.

Input Parameters

- **transa**
  Specifies the operation.
  
  If \( transa = 'N' \) or \( 'n' \), then \( y := \alpha \cdot \text{inv}(A) \cdot x \)
  
  If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha \cdot \text{inv}(A^T) \cdot x, \)

- **m**
  Number of columns of the matrix \( A \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **matdescra**
  Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**
  Array containing non-zero elements of the matrix \( A \).
  
  Its length is \( \text{pntr}[m-1] - \text{pntr}[0] \).
  
  Refer to values array description in CSC Format for more details.

**NOTE**

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.
For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix $A$.

For zero-based indexing, array containing the row indices for each non-zero element of the matrix $A$.

Its length is equal to length of the $val$ array.

Refer to columns array description in CSC Format for more details.

**NOTE**

Row indices must be sorted in increasing order for each column.

Array of length $m$.

This array contains column indices, such that $pntrb[i] - pntrb[0]$ is the first index of column $i$ in the arrays $val$ and $indx$.

Refer to pointerb array description in CSC Format for more details.

Array of length $m$.

This array contains column indices, such that $pntrb[i] - pntrb[0] - 1$ is the last index of column $i$ in the arrays $val$ and $indx$.

Refer to pointerE array description in CSC Format for more details.

Array, size at least $m$.

On entry, the array $x$ must contain the vector $x$. The elements are accessed with unit increment.

Array, size at least $m$.

On entry, the array $y$ must contain the vector $y$. The elements are accessed with unit increment.

**Output Parameters**

Contains the solution vector $x$.

**mkl_?coosv**

Solves a system of linear equations for a sparse matrix in the coordinate format (deprecated).

**Syntax**

```c
void mkl_scoosv (const char *transa , const MKL_INT *m , const float *alpha , const char *matdescra , const float *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const float *x , float *y );
void mkl_dcoosv (const char *transa , const MKL_INT *m , const double *alpha , const char *matdescra , const double *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const double *x , double *y );
void mkl_ccoosv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT *rowind , const MKL_INT *colind , const MKL_INT *nnz , const MKL_Complex8 *x , MKL_Complex8 *y );
```
void mkl_zcoosv (const char *transa, const MKL_INT *m, const MKL_Complex16 *alpha, 
const char *matdescra, const MKL_Complex16 *val, const MKL_INT *rowind, const 
MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *x, MKL_Complex16 *y );

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS 
interface instead.

The mkl _?coosv routine solves a system of linear equations with matrix-vector operations for a sparse 
matrix in the coordinate format:

\[ y := \alpha \cdot \text{inv}(A) \cdot x \]

or

\[ y := \alpha \cdot \text{inv}(A^T) \cdot x , \]

where:
\( \alpha \) is scalar, \( x \) and \( y \) are vectors, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main 
diagonal, \( A^T \) is the transpose of \( A \).

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

**transa**

Specifies the system of linear equations.

If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha \cdot \text{inv}(A) \cdot x \)

If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha \cdot \text{inv}(A^T) \) \cdot x , \)

\m

Number of rows of the matrix \( A \).

\alpha

Array of six elements, specifies properties of the matrix used for operation. 
Only first four array elements are used, their possible values are given in 
Table "Possible Values of the Parameter matdescra (descra)". Possible 
combinations of element values of this parameter are given in Table 
"Possible Combinations of Element Values of the Parameter matdescra".

\val

Array of length \( nnz \), contains non-zero elements of the matrix \( A \) in the 
arbitrary order.

Refer to values array description in Coordinate Format for more details.

\rowind

Array of length \( nnz \).

For one-based indexing, contains the row indices plus one for each non-zero 
element of the matrix \( A \).

For zero-based indexing, contains the row indices for each non-zero 
element of the matrix \( A \).

188
Refer to \textit{rows} array description in \textit{Coordinate Format} for more details.

\textit{colind}

Array of length \textit{nnz}.

For one-based indexing, contains the column indices plus one for each non-zero element of the matrix \(A\).

For zero-based indexing, contains the column indices for each non-zero element of the matrix \(A\).

Refer to \textit{columns} array description in \textit{Coordinate Format} for more details.

\textit{nnz}

Specifies the number of non-zero element of the matrix \(A\).

Refer to \textit{nnz} description in \textit{Coordinate Format} for more details.

\textit{x}

Array, size at least \(m\).

On entry, the array \(x\) must contain the vector \(x\). The elements are accessed with unit increment.

\textit{y}

Array, size at least \(m\).

On entry, the array \(y\) must contain the vector \(y\). The elements are accessed with unit increment.

\textbf{Output Parameters}

\textit{y}

Contains solution vector \(x\).

\textbf{mkl\_?csrmm}

\textit{Computes matrix - matrix product of a sparse matrix stored in the CSR format (deprecated).}

\textbf{Syntax}

\begin{verbatim}
void mkl_scsrmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *b, const MKL_INT *ldb, const float *beta, float *c, const MKL_INT *ldc);
void mkl_dcsrmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *b, const MKL_INT *ldb, const double *beta, double *c, const MKL_INT *ldc);
void mkl_ccsrmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *b, const MKL_INT *ldb, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ldc);
void mkl_zcsrmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *b, const MKL_INT *ldb, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ldc);
\end{verbatim}
Include Files

• mkl.h

Description

This routine is deprecated. Use Use mkl_sparse_?_mm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmm routine performs a matrix-matrix operation defined as

\[ C := \alpha A B + \beta C \]

or

\[ C := \alpha A^T B + \beta C \]

or

\[ C := \alpha A^H B + \beta C, \]

where:

\( \alpha \) and \( \beta \) are scalars,

\( B \) and \( C \) are dense matrices, \( A \) is an \( m \)-by-\( k \) sparse matrix in compressed sparse row (CSR) format, \( A^T \) is the transpose of \( A \), and \( A^H \) is the conjugate transpose of \( A \).

**NOTE**

This routine supports a CSR format both with one-based indexing and zero-based indexing.

Input Parameters

**transa**

Specifies the operation.

If \( \text{transa} = 'N' \) or \( 'n' \), then \( C := \alpha A B + \beta C \),

If \( \text{transa} = 'T' \) or \( 't' \), then \( C := \alpha A^T B + \beta C \),

If \( \text{transa} = 'C' \) or \( 'c' \), then \( C := \alpha A^H B + \beta C \).

**m**

Number of rows of the matrix \( A \).

**n**

Number of columns of the matrix \( C \).

**k**

Number of columns of the matrix \( A \).

**alpha**

Specifies the scalar \( \alpha \).

**matdescra**

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table “Possible Values of the Parameter matdescra (descra)”. Possible combinations of element values of this parameter are given in Table “Possible Combinations of Element Values of the Parameter matdescra”.

**val**

Array containing non-zero elements of the matrix \( A \).

For zero-based indexing its length is \( \text{pntre}[m-1] - \text{pntrb}[0] \).

Refer to values array description in CSR Format for more details.
For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix $A$.

For zero-based indexing, array containing the column indices for each non-zero element of the matrix $A$.

Its length is equal to length of the $val$ array.

Refer to $columns$ array description in CSR Format for more details.

Array of length $m$.

This array contains row indices, such that $pntrb[I] - pntrb[0]$ is the first index of row $I$ in the arrays $val$ and $indx$.

Refer to $pointerb$ array description in CSR Format for more details.

Array of length $m$.

This array contains row indices, such that $pntre[I] - pntrb[0] - 1$ is the last index of row $I$ in the arrays $val$ and $indx$.

Refer to $pointerE$ array description in CSR Format for more details.

Array, size $ldb$ by at least $n$ for non-transposed matrix $A$ and at least $m$ for transposed for one-based indexing, and (at least $k$ for non-transposed matrix $A$ and at least $m$ for transposed, $ldb$) for zero-based indexing.

On entry with $transa='N'$ or 'n', the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$, otherwise the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

Specifies the leading dimension of $b$ for one-based indexing, and the second dimension of $b$ for zero-based indexing, as declared in the calling (sub)program.

Specifies the scalar $beta$.

Array, size $ldc$ by $n$ for one-based indexing, and ($m$, $ldc$) for zero-based indexing.

On entry, the leading $m$-by-$n$ part of the array $c$ must contain the matrix $C$, otherwise the leading $k$-by-$n$ part of the array $c$ must contain the matrix $C$.

Specifies the leading dimension of $c$ for one-based indexing, and the second dimension of $c$ for zero-based indexing, as declared in the calling (sub)program.

Overwritten by the matrix $(alpha*\mathbf{A}*\mathbf{B} + beta* \mathbf{C})$, $(alpha*\mathbf{A}^T*\mathbf{B} + beta* \mathbf{C})$, or $(alpha*\mathbf{A}^H*\mathbf{B} + beta* \mathbf{C})$.

$\texttt{mkl}_\texttt{-}bsrmm$

Computes matrix - matrix product of a sparse matrix stored in the BSR format (deprecated).
Syntax

void mkl_sbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_INT *lb , const float *alpha , const char *matdescra , const float *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const float *b , const MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );

void mkl_dbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_INT *lb , const double *alpha , const char *matdescra , const double *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const double *b , const MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );

void mkl_cbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_INT *lb , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zbsrmm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_INT *lb , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT *indx , const MKL_INT *pntrb , const MKL_INT *pntre , const MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ldc );

Include Files
- mkl.h

Description

This routine is deprecated. Use Use mkl_sparse_?_mm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrmm routine performs a matrix-matrix operation defined as

\[ C := \alpha A \cdot B + \beta C \]

or

\[ C := \alpha A^\top \cdot B + \beta C \]

or

\[ C := \alpha A^\mathbb{H} \cdot B + \beta C , \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( B \) and \( C \) are dense matrices, \( A \) is an \( m \)-by-\( k \) sparse matrix in block sparse row (BSR) format, \( A^\top \) is the transpose of \( A \), and \( A^\mathbb{H} \) is the conjugate transpose of \( A \).

**NOTE**

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

- \( \text{transa} \): Specifies the operation.
If $\text{transa} = 'N'$ or '$n'$, then the matrix-matrix product is computed as 
\[
C := \alpha \cdot A \cdot B + \beta \cdot C
\]

If $\text{transa} = 'T'$ or '$t'$, then the matrix-vector product is computed as 
\[
C := \alpha \cdot A^T \cdot B + \beta \cdot C
\]

If $\text{transa} = 'C'$ or '$c'$, then the matrix-vector product is computed as 
\[
C := \alpha \cdot A^H \cdot B + \beta \cdot C,
\]

$m$
Number of block rows of the matrix $A$.

$n$
Number of columns of the matrix $C$.

$k$
Number of block columns of the matrix $A$.

$lb$
Size of the block in the matrix $A$.

$\alpha$
Specifies the scalar $\alpha$.

$\text{matdescra}$
Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter $\text{matdescra (descra)}$". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter $\text{matdescra}$".

$val$
Array containing elements of non-zero blocks of the matrix $A$. Its length is equal to the number of non-zero blocks in the matrix $A$ multiplied by $lb \cdot lb$. Refer to the $values$ array description in $BSR$ Format for more details.

$indx$
For one-based indexing, array containing the column indices plus one for each non-zero block in the matrix $A$.

For zero-based indexing, array containing the column indices for each non-zero block in the matrix $A$.

Its length is equal to the number of non-zero blocks in the matrix $A$. Refer to the $columns$ array description in $BSR$ Format for more details.

$\text{pntrb}$
Array of length $m$.

This array contains row indices, such that $\text{pntrb}[I] - \text{pntrb}[0]$ is the first index of block row $I$ in the array $indx$.

Refer to $pointerB$ array description in $BSR$ Format for more details.

$\text{pntrc}$
Array of length $m$.

This array contains row indices, such that $\text{pntrc}[I] - \text{pntrc}[0] - 1$ is the last index of block row $I$ in the array $indx$.

Refer to $pointerE$ array description in $BSR$ Format for more details.

$b$
Array, size $ldb$ by at least $n$ for non-transposed matrix $A$ and at least $m$ for transposed for one-based indexing, and (at least $k$ for non-transposed matrix $A$ and at least $m$ for transposed, $ldb$) for zero-based indexing.

On entry with $\text{transa}='N'$ or '$n$', the leading $n$-by-$k$ block part of the array $b$ must contain the matrix $B$, otherwise the leading $m$-by-$n$ block part of the array $b$ must contain the matrix $B$. 
**ldb**

Specifies the leading dimension (in blocks) of \( b \) as declared in the calling (sub)program.

**beta**

Specifies the scalar \( \beta \).

**c**

Array, size \( ldc \times n \) for one-based indexing, size \( k \times ldc \) for zero-based indexing.

On entry, the leading \( m \)-by-\( n \) block part of the array \( c \) must contain the matrix \( C \), otherwise the leading \( n \)-by-\( k \) block part of the array \( c \) must contain the matrix \( C \).

**ldc**

Specifies the leading dimension (in blocks) of \( c \) as declared in the calling (sub)program.

**Output Parameters**

**c**

Overwritten by the matrix \((\alpha A^T B + \beta C)\) or \((\alpha A^H B + \beta C)\) or \((\alpha A^* B + \beta C)\).

---

**mkl_?cscmm**

Computes matrix-matrix product of a sparse matrix stored in the CSC format (deprecated).

**Syntax**

```c
void mkl_scscmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *b, const MKL_INT *ldb, const float *beta, float *c, const MKL_INT *ldc);
void mkl_dcscmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *b, const MKL_INT *ldb, const double *beta, double *c, const MKL_INT *ldc);
void mkl_ccscmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *b, const MKL_INT *ldb, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ldc);
void mkl_zcscmm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *b, const MKL_INT *ldb, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ldc);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use Use mkl_sparse_?_mm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.
The `mkl_?cscmm` routine performs a matrix-matrix operation defined as

\[ C := \alpha A B + \beta C \]

or

\[ C := \alpha A^T B + \beta C, \]

or

\[ C := \alpha A^H B + \beta C, \]

where:

\( \alpha \) and \( \beta \) are scalars,

\( B \) and \( C \) are dense matrices, \( A \) is an \( m \)-by-\( k \) sparse matrix in compressed sparse column (CSC) format, \( A^T \) is the transpose of \( A \), and \( A^H \) is the conjugate transpose of \( A \).

**NOTE**

This routine supports CSC format both with one-based indexing and zero-based indexing.

## Input Parameters

- **transa**: Specifies the operation.
  - If `transa = 'N'` or `'n'`, then \( C := \alpha A B + \beta C \)
  - If `transa = 'T'` or `'t'`, then \( C := \alpha A^T B + \beta C \)
  - If `transa = 'C'` or `'c'`, then \( C := \alpha A^H B + \beta C \)

- **m**: Number of rows of the matrix \( A \).

- **n**: Number of columns of the matrix \( C \).

- **k**: Number of columns of the matrix \( A \).

- **alpha**: Specifies the scalar \( \alpha \).

- **matdescra**: Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**: Array containing non-zero elements of the matrix \( A \).
  - Its length is \( \text{pntrb}[k-1] - \text{pntrb}[0] \).
  - Refer to `values` array description in CSC Format for more details.

- **indx**: For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix \( A \).
  - For zero-based indexing, array containing the column indices for each non-zero element of the matrix \( A \).
  - Its length is equal to length of the `val` array.
  - Refer to `rows` array description in CSC Format for more details.

- **pntrb**: Array of length \( k \).
This array contains column indices, such that \( \text{pntrb}[i] - \text{pntrb}[0] \) is the first index of column \( i \) in the arrays \( \text{val} \) and \( \text{indx} \).

Refer to \( \text{pointerb} \) array description in \textit{CSC Format} for more details.

\textit{pntre} Array of length \( k \).

This array contains column indices, such that \( \text{pntre}[i] - \text{pntrb}[0] - 1 \) is the last index of column \( i \) in the arrays \( \text{val} \) and \( \text{indx} \).

Refer to \( \text{pointerE} \) array description in \textit{CSC Format} for more details.

\textit{b} Array, size \( \text{ldb} \) by at least \( n \) for non-transposed matrix \( A \) and at least \( m \) for transposed for one-based indexing, and (at least \( k \) for non-transposed matrix \( A \) and at least \( m \) for transposed, \( \text{ldb} \)) for zero-based indexing.

On entry with \( \text{transa} = 'N' \) or \( 'n' \), the leading \( k \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \), otherwise the leading \( m \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \).

\textit{ldb} Specifies the leading dimension of \( b \) for one-based indexing, and the second dimension of \( b \) for zero-based indexing, as declared in the calling (sub)program.

\textit{beta} Specifies the scalar \( \beta \).

\textit{c} Array, size \( \text{ldc} \) by \( n \) for one-based indexing, and \( (m, \text{ldc}) \) for zero-based indexing.

On entry, the leading \( m \)-by-\( n \) part of the array \( c \) must contain the matrix \( C \), otherwise the leading \( k \)-by-\( n \) part of the array \( c \) must contain the matrix \( C \).

\textit{ldc} Specifies the leading dimension of \( c \) for one-based indexing, and the second dimension of \( c \) for zero-based indexing, as declared in the calling (sub)program.

\textbf{Output Parameters}

\textit{c} Overwritten by the matrix \((\alpha A B + \beta C)\) or \((\alpha A^T B + \beta C)\) or \((\alpha A^H B + \beta C)\).

\textbf{mkl_?coomm}

\textit{Computes matrix-matrix product of a sparse matrix stored in the coordinate format (deprecated).}

\textbf{Syntax}

\begin{verbatim}
void mkl_scoomm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const float *b, const MKL_INT *ldb, const float *beta, float *c, const MKL_INT *ldc);

void mkl_dcoomm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const double *b, const MKL_INT *ldb, const double *beta, double *c, const MKL_INT *ldc);
\end{verbatim}
void mkl_ccoomm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex8 *b, const MKL_INT *ldb, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ldc);

void mkl_zcoomm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *b, const MKL_INT *ldb, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ldc);

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?coomm routine performs a matrix-matrix operation defined as

\[ C := \alpha A B + \beta C \]

or

\[ C := \alpha A^T B + \beta C, \]

or

\[ C := \alpha A^H B + \beta C, \]

where:
\( \alpha \) and \( \beta \) are scalars,

\( B \) and \( C \) are dense matrices, \( A \) is an \( m \)-by-\( k \) sparse matrix in the coordinate format, \( A^T \) is the transpose of \( A \), and \( A^H \) is the conjugate transpose of \( A \).

NOTE
This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

transa

Specifies the operation.

If \( \text{transa} = \text{'N'} \) or \( \text{'n'} \), then \( C := \alpha A B + \beta C \)

If \( \text{transa} = \text{'T'} \) or \( \text{'t'} \), then \( C := \alpha A^T B + \beta C, \)

If \( \text{transa} = \text{'C'} \) or \( \text{'c'} \), then \( C := \alpha A^H B + \beta C. \)

m

Number of rows of the matrix \( A \).

n

Number of columns of the matrix \( C \).

k

Number of columns of the matrix \( A \).

alpha

Specifies the scalar \( \alpha \).
matdescra

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

val

Array of length \( nnz \), contains non-zero elements of the matrix \( A \) in the arbitrary order.

Refer to values array description in Coordinate Format for more details.

rowind

Array of length \( nnz \).

For one-based indexing, contains the row indices plus one for each non-zero element of the matrix \( A \).

For zero-based indexing, contains the row indices for each non-zero element of the matrix \( A \).

Refer to rows array description in Coordinate Format for more details.

colind

Array of length \( nnz \).

For one-based indexing, contains the column indices plus one for each non-zero element of the matrix \( A \).

For zero-based indexing, contains the column indices for each non-zero element of the matrix \( A \).

Refer to columns array description in Coordinate Format for more details.

nnz

Specifies the number of non-zero element of the matrix \( A \).

Refer to \( nnz \) description in Coordinate Format for more details.

b

Array, size \( ldb \) by at least \( n \) for non-transposed matrix \( A \) and at least \( m \) for transposed for one-based indexing, and (at least \( k \) for non-transposed matrix \( A \) and at least \( m \) for transposed, \( ldb \)) for zero-based indexing.

On entry with \( transa = 'N' \) or \( 'n' \), the leading \( k \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \), otherwise the leading \( m \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \).

ldb

Specifies the leading dimension of \( b \) for one-based indexing, and the second dimension of \( b \) for zero-based indexing, as declared in the calling (sub)program.

beta

Specifies the scalar \( beta \).

c

Array, size \( ldc \) by \( n \) for one-based indexing, and \( (m, \ ldc) \) for zero-based indexing.

On entry, the leading \( m \)-by-\( n \) part of the array \( c \) must contain the matrix \( C \), otherwise the leading \( k \)-by-\( n \) part of the array \( c \) must contain the matrix \( C \).

ldc

Specifies the leading dimension of \( c \) for one-based indexing, and the second dimension of \( c \) for zero-based indexing, as declared in the calling (sub)program.
Output Parameters

\[ c \quad \text{Overwritten by the matrix } (\alpha A B + \beta C), (\alpha A^T B + \beta C), \text{ or } (\alpha A^H B + \beta C). \]

mkl_?csrsm

Solves a system of linear matrix equations for a sparse matrix in the CSR format (deprecated).

Syntax

\[
\begin{align*}
\text{void mkl_scsrsm} & \quad (\text{const char}*\text{transa}, \text{const MKL_INT}*m, \text{const MKL_INT}*n, \text{const float}*\alpha, \text{const char}*\text{matdescra}, \text{const float}*\text{val}, \text{const MKL_INT}^{*}\text{indx}, \text{const MKL_INT}^{*}\text{pntrb}, \text{const MKL_INT}^{*}\text{pntre}, \text{const float}*b, \text{const MKL_INT}^{*}\text{ldb}, \text{float}*c, \text{const MKL_INT}^{*}\text{ldc}); \\
\text{void mkl_dcsrsm} & \quad (\text{const char}*\text{transa}, \text{const MKL_INT}*m, \text{const MKL_INT}*n, \text{const double}*\alpha, \text{const char}*\text{matdescra}, \text{const double}*\text{val}, \text{const MKL_INT}^{*}\text{indx}, \text{const MKL_INT}^{*}\text{pntrb}, \text{const MKL_INT}^{*}\text{pntre}, \text{const double}*b, \text{const MKL_INT}^{*}\text{ldb}, \text{double}*c, \text{const MKL_INT}^{*}\text{ldc}); \\
\text{void mkl_ccsrsm} & \quad (\text{const char}*\text{transa}, \text{const MKL_INT}*m, \text{const MKL_INT}*n, \text{const MKL_Complex8}*\alpha, \text{const char}*\text{matdescra}, \text{const MKL_Complex8}*\text{val}, \text{const MKL_INT}^{*}\text{indx}, \text{const MKL_INT}^{*}\text{pntrb}, \text{const MKL_INT}^{*}\text{pntre}, \text{const MKL_Complex8}*b, \text{const MKL_INT}^{*}\text{ldb}, \text{MKL_Complex8}*c, \text{const MKL_INT}^{*}\text{ldc}); \\
\text{void mkl_zcsrsm} & \quad (\text{const char}*\text{transa}, \text{const MKL_INT}*m, \text{const MKL_INT}*n, \text{const MKL_Complex16}*\alpha, \text{const char}*\text{matdescra}, \text{const MKL_Complex16}*\text{val}, \text{const MKL_INT}^{*}\text{indx}, \text{const MKL_INT}^{*}\text{pntrb}, \text{const MKL_INT}^{*}\text{pntre}, \text{const MKL_Complex16}*b, \text{const MKL_INT}^{*}\text{ldb}, \text{MKL_Complex16}*c, \text{const MKL_INT}^{*}\text{ldc});
\end{align*}
\]

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrsm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSR format:

\[
C := \alpha \text{inv}(A)B
\]

or

\[
C := \alpha \text{inv}(A^T)B,
\]

where:

\( \alpha \) is scalar, \( B \) and \( C \) are dense matrices, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

NOTE

This routine supports a CSR format both with one-based indexing and zero-based indexing.
Input Parameters

transa

Specifies the system of linear equations.

If transa = 'N' or 'n', then \( C := \alpha \cdot \text{inv}(A) \cdot B \)

If transa = 'T' or 't' or 'C' or 'c', then \( C := \alpha \cdot \text{inv}(A^T) \cdot B \)

m

Number of columns of the matrix \( A \).

n

Number of columns of the matrix \( C \).

alpha

Specifies the scalar \( \alpha \).

matdescra

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

val

Array containing non-zero elements of the matrix \( A \).

For zero-based indexing its length is \( pntre[m-1] - pntrb[0] \).

Refer to values array description in CSR Format for more details.

NOTE

The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix \( A \).

For zero-based indexing, array containing the column indices for each non-zero element of the matrix \( A \).

Its length is equal to length of the val array.

Refer to columns array description in CSR Format for more details.

NOTE

Column indices must be sorted in increasing order for each row.

pntrb

Array of length \( m \).

This array contains row indices, such that \( \text{pntrb}[i] - \text{pntrb}[0] \) is the first index of row \( i \) in the arrays \( \text{val} \) and \( \text{indx} \).

Refer to pointerb array description in CSR Format for more details.

pntre

Array of length \( m \).

For zero-based indexing this array contains row indices, such that \( \text{pntre}[i] - \text{pntrb}[0] - 1 \) is the last index of row \( i \) in the arrays \( \text{val} \) and \( \text{indx} \).
Refer to pointerE array description in CSR Format for more details.

*b*

Array, size ldb* n for one-based indexing, and (m, ldb) for zero-based indexing.

On entry the leading m-by-n part of the array b must contain the matrix B.

*ldb*

Specifies the leading dimension of b for one-based indexing, and the second dimension of b for zero-based indexing, as declared in the calling (sub)program.

*ldc*

Specifies the leading dimension of c for one-based indexing, and the second dimension of c for zero-based indexing, as declared in the calling (sub)program.

**Output Parameters**

*c*

Array, size ldc by n for one-based indexing, and (m, ldc) for zero-based indexing.

The leading m-by-n part of the array c contains the output matrix C.

**mkl_?cscsm**

Solves a system of linear matrix equations for a sparse matrix in the CSC format (deprecated).

**Syntax**

```c
void mkl_scscsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *b, const MKL_INT *ldb, float *c, const MKL_INT *ldc);
void mkl_dcscsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *b, const MKL_INT *ldb, double *c, const MKL_INT *ldc);
void mkl_ccscsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *b, const MKL_INT *ldb, MKL_Complex8 *c, const MKL_INT *ldc);
void mkl_zcscsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *b, const MKL_INT *ldb, MKL_Complex16 *c, const MKL_INT *ldc);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_trsm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.
The mkl_?cscsm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the CSC format:

\[ C := \alpha \cdot \text{inv}(A) \cdot B \]

or

\[ C := \alpha \cdot \text{inv}(A^T) \cdot B, \]

where:

\( \alpha \) is scalar, \( B \) and \( C \) are dense matrices, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports a CSC format both with one-based indexing and zero-based indexing.

**Input Parameters**

- **transa**
  Specifies the system of equations.
  - If \( \text{transa} = 'N' \) or \'n' \), then \( C := \alpha \cdot \text{inv}(A) \cdot B \)
  - If \( \text{transa} = 'T' \) or \'t' \) or \'C' \) or \'c' \), then \( C := \alpha \cdot \text{inv}(A^T) \cdot B, \)

- **m**
  Number of columns of the matrix \( A \).

- **n**
  Number of columns of the matrix \( C \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **matdescra**
  Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**
  Array containing non-zero elements of the matrix \( A \).
  For zero-based indexing its length is \( \text{pnre}[m] - \text{pntrb}[0] \).
  Refer to values array description in CSC Format for more details.

**NOTE**
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

- **indx**
  For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix \( A \). For zero-based indexing, array containing the row indices for each non-zero element of the matrix \( A \).
  Refer to rows array description in CSC Format for more details.
Row indices must be sorted in increasing order for each column.

**pntrb**

Array of length \( m \).
This array contains column indices, such that \( pntrb[I] - pntrb[0] \) is the first index of column \( I \) in the arrays \( val \) and \( indx \).
Refer to \( pointerb \) array description in **CSC Format** for more details.

**pntrE**

Array of length \( m \).
This array contains column indices, such that \( pntrE[I] - pntrb[I]-1 \) is the last index of column \( I \) in the arrays \( val \) and \( indx \).
Refer to \( pointerE \) array description in **CSC Format** for more details.

**b**

Array, size \( ldb \) by \( n \) for one-based indexing, and \((m, ldb)\) for zero-based indexing.
On entry the leading \( m \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \).

**ldb**

Specifies the leading dimension of \( b \) for one-based indexing, and the second dimension of \( b \) for zero-based indexing, as declared in the calling (sub)program.

**ldc**

Specifies the leading dimension of \( c \) for one-based indexing, and the second dimension of \( c \) for zero-based indexing, as declared in the calling (sub)program.

**Output Parameters**

**c**

Array, size \( ldc \) by \( n \) for one-based indexing, and \((m, ldc)\) for zero-based indexing.
The leading \( m \)-by-\( n \) part of the array \( c \) contains the output matrix \( C \).

**mkl_?coosm**

Solves a system of linear matrix equations for a sparse matrix in the coordinate format (deprecated).

**Syntax**

void mkl_scoosm (const char *transa, const MKL_INT *m, const MKL_INT *n, const float *alpha, const char *matdesca, const float *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const float *b, const MKL_INT *ldb, float *c, const MKL_INT *ldc);

void mkl_dcoosm (const char *transa, const MKL_INT *m, const MKL_INT *n, const double *alpha, const char *matdesca, const double *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const double *b, const MKL_INT *ldb, double *c, const MKL_INT *ldc);

void mkl_ccoosm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const char *matdesca, const MKL_Complex8 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex8 *b, const MKL_INT *ldb, MKL_Complex8 *c, const MKL_INT *ldc);
void mkl_zcoosm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *rowind, const MKL_INT *colind, const MKL_INT *nnz, const MKL_Complex16 *b, const MKL_INT *ldb, MKL_Complex16 *c, const MKL_INT *ldc);

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_zcoosm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the coordinate format:

\[ C := alpha \cdot \text{inv}(A) \cdot B \]

or

\[ C := alpha \cdot \text{inv}(A^T) \cdot B, \]

where:

- \( alpha \) is scalar, \( B \) and \( C \) are dense matrices, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports a coordinate format both with one-based indexing and zero-based indexing.

Input Parameters

- **transa**: Specifies the system of linear equations.
  - If \( transa = 'N' \) or \( 'n' \), then the matrix-matrix product is computed as \( C := alpha \cdot \text{inv}(A) \cdot B \).
  - If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then the matrix-vector product is computed as \( C := alpha \cdot \text{inv}(A^T) \cdot B \).

- **m**: Number of rows of the matrix \( A \).

- **n**: Number of columns of the matrix \( C \).

- **alpha**: Specifies the scalar \( alpha \).

- **matdescra**: Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

- **val**: Array of length \( nnz \), contains non-zero elements of the matrix \( A \) in the arbitrary order.
  - Refer to \( values \) array description in Coordinate Format for more details.

- **rowind**

- **colind**

- **nnz**

- **b**: Constant dense matrix \( B \).

- **ldb**

- **ldc**

- **c**: Constant dense matrix \( C \).

- **alpha**: Constant scalar \( alpha \).
For one-based indexing, contains the row indices plus one for each non-zero element of the matrix $A$.
For zero-based indexing, contains the row indices for each non-zero element of the matrix $A$.

Refer to rows array description in Coordinate Format for more details.

$\text{colind}$

Array of length $nnz$.
For one-based indexing, contains the column indices plus one for each non-zero element of the matrix $A$.
For zero-based indexing, contains the row indices for each non-zero element of the matrix $A$.

Refer to columns array description in Coordinate Format for more details.

$nnz$

Specifies the number of non-zero element of the matrix $A$.
Refer to nnz description in Coordinate Format for more details.

$b$

Array, size $ldb$ by $n$ for one-based indexing, and $(m, ldb)$ for zero-based indexing.
Before entry the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

$ldb$

Specifies the leading dimension of $b$ for one-based indexing, and the second dimension of $b$ for zero-based indexing, as declared in the calling (sub)program.

$ldc$

Specifies the leading dimension of $c$ for one-based indexing, and the second dimension of $c$ for zero-based indexing, as declared in the calling (sub)program.

**Output Parameters**

$c$

Array, size $ldc$ by $n$ for one-based indexing, and $(m, ldc)$ for zero-based indexing.
The leading $m$-by-$n$ part of the array $c$ contains the output matrix $C$.

**mkl\_?bsrsm**

Solves a system of linear matrix equations for a sparse matrix in the BSR format (deprecated).

**Syntax**

```c
void mkl_sbsrsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *lb, const float *alpha, const char *matdescra, const float *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const float *b, const MKL_INT *ldb, float *c, const MKL_INT *ldc);
void mkl_dbsrsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *lb, const double *alpha, const char *matdescra, const double *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const double *b, const MKL_INT *ldb, double *c, const MKL_INT *ldc);
```
void mkl_cbsrsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *lb, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex8 *b, const MKL_INT *ldb, MKL_Complex8 *c, const MKL_INT *ldc);

void mkl_zbsrsm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *lb, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *indx, const MKL_INT *pntrb, const MKL_INT *pntre, const MKL_Complex16 *b, const MKL_INT *ldb, MKL_Complex16 *c, const MKL_INT *ldc);

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?bsrsm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the BSR format:

\[
C := \alpha * \text{inv}(A) * B
\]

or

\[
C := \alpha * \text{inv}(A^T) * B,
\]

where:

\(\alpha\) is scalar, \(B\) and \(C\) are dense matrices, \(A\) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \(A^T\) is the transpose of \(A\).

NOTE

This routine supports a BSR format both with one-based indexing and zero-based indexing.

Input Parameters

**transa**

Specifies the operation.

- If \(\text{transa} = 'N'\) or \(\text{'n'}\), then the matrix-matrix product is computed as \(C := \alpha * \text{inv}(A) * B\).
- If \(\text{transa} = 'T'\) or \(\text{'t'}\) or \(\text{'C'}\) or \(\text{'c'}\), then the matrix-vector product is computed as \(C := \alpha * \text{inv}(A^T) * B\).

**m**

Number of block columns of the matrix \(A\).

**n**

Number of columns of the matrix \(C\).

**lb**

Size of the block in the matrix \(A\).

**alpha**

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**matdescra**

Specifies the scalar \(\alpha\).
val

Array containing elements of non-zero blocks of the matrix $A$. Its length is equal to the number of non-zero blocks in the matrix $A$ multiplied by $lb*lb$. Refer to the values array description in BSR Format for more details.

NOTE
The non-zero elements of the given row of the matrix must be stored in the same order as they appear in the row (from left to right).

No diagonal element can be omitted from a sparse storage if the solver is called with the non-unit indicator.

indx

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix $A$. For zero-based indexing, array containing the column indices for each non-zero element of the matrix $A$. Its length is equal to the number of non-zero blocks in the matrix $A$. Refer to the columns array description in BSR Format for more details.

pntrb

Array of length $m$.

This array contains row indices, such that $\text{pntrb}[i] - \text{pntrb}[0]$ is the first index of block row $i$ in the array $\text{indx}$.

Refer to pointerB array description in BSR Format for more details.

pnre

Array of length $m$.

This array contains row indices, such that $\text{pnre}[i] - \text{pntrb}[0] - 1$ is the last index of block row $i$ in the arrays $\text{val}$ and $\text{indx}$.

Refer to pointerE array description in BSR Format for more details.

b

Array, size $ldb* n$ for one-based indexing, size $m* ldb$ for zero-based indexing.

On entry the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

$ldb$

Specifies the leading dimension (in blocks) of $b$ as declared in the calling (sub)program.

$ldc$

Specifies the leading dimension (in blocks) of $c$ as declared in the calling (sub)program.

Output Parameters

$c$

Array, size $ldc* n$ for one-based indexing, size $m* ldc$ for zero-based indexing.

The leading $m$-by-$n$ part of the array $c$ contains the output matrix $C$.

mkl_?diamv

Computes matrix - vector product for a sparse matrix in the diagonal format with one-based indexing (deprecated).
Syntax

```c
void mkl_sdiamv (const char *transa, const MKL_INT *m, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const float *x, const float *beta, float *y);
void mkl_ddiamv (const char *transa, const MKL_INT *m, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const double *x, const double *beta, double *y);
void mkl_cdiamv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex8 *x, const MKL_Complex8 *beta, MKL_Complex8 *y);
void mkl_zdiamv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);
```

Include Files

- mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_?diamv` routine performs a matrix-vector operation defined as

\[
y := \alpha A x + \beta y
\]

or

\[
y := \alpha A^T x + \beta y,
\]

where:

- \(\alpha\) and \(\beta\) are scalars,
- \(x\) and \(y\) are vectors,
- \(A\) is an \(m\)-by-\(k\) sparse matrix stored in the diagonal format, \(A^T\) is the transpose of \(A\).

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

- **transa**: Specifies the operation.
  
  If `transa` = 'N' or 'n', then \(y := \alpha A x + \beta y\),
  
  If `transa` = 'T' or 't' or 'C' or 'c', then \(y := \alpha A^T x + \beta y\).

- **m**: Number of rows of the matrix \(A\).
- **k**: Number of columns of the matrix \(A\).
alpha

Specifies the scalar $alpha$.

matdescra

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

val

Two-dimensional array of size $lval$ by $ndiag$, contains non-zero diagonals of the matrix $A$. Refer to values array description in Diagonal Storage Scheme for more details.

lval

Leading dimension of $val$, $lval \geq m$. Refer to $lval$ description in Diagonal Storage Scheme for more details.

idiag

Array of length $ndiag$, contains the distances between main diagonal and each non-zero diagonals in the matrix $A$. Refer to distance array description in Diagonal Storage Scheme for more details.

ndiag

Specifies the number of non-zero diagonals of the matrix $A$.

x

Array, size at least $k$ if $transa = 'N'$ or 'n', and at least $m$ otherwise. On entry, the array $x$ must contain the vector $x$.

beta

Specifies the scalar $beta$.

y

Array, size at least $m$ if $transa = 'N'$ or 'n', and at least $k$ otherwise. On entry, the array $y$ must contain the vector $y$.

Output Parameters

y

Overwritten by the updated vector $y$.

mkl_?skymv

Computes matrix - vector product for a sparse matrix in the skyline storage format with one-based indexing (deprecated).

Syntax

```
void mkl_sskymv (const char *transa, const MKL_INT *m, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *pntr, const float *x, const float *beta, float *y);
void mkl_dskymv (const char *transa, const MKL_INT *m, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *pntr, const double *x, const double *beta, double *y);
void mkl_cskymv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *pntr, const MKL_Complex8 *x, const MKL_Complex8 *beta, MKL_Complex8 *y);
void mkl_zskymv (const char *transa, const MKL_INT *m, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *pntr, const MKL_Complex16 *x, const MKL_Complex16 *beta, MKL_Complex16 *y);
```
Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_mv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?skymv routine performs a matrix-vector operation defined as

\[ y := \alpha A x + \beta y \]

or

\[ y := \alpha A^T x + \beta y, \]

where:
\( \alpha \) and \( \beta \) are scalars,
\( x \) and \( y \) are vectors,
\( A \) is an \( m \)-by-\( k \) sparse matrix stored using the skyline storage scheme, \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only one-based indexing of the input arrays.

Input Parameters

\( transa \)
Specifies the operation.
If \( transa = 'N' \) or \( 'n' \), then \( y := \alpha A x + \beta y \)
If \( transa = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha A^T x + \beta y \),

\( m \)
Number of rows of the matrix \( A \).

\( k \)
Number of columns of the matrix \( A \).

\( alpha \)
Specifies the scalar \( \alpha \).

\( matdescra \)
Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**NOTE**
General matrices (\( matdescra[0]='G' \)) is not supported.

\( val \)
Array containing the set of elements of the matrix \( A \) in the skyline profile form.
If \( matdescra[1]='L' \), then \( val \) contains elements from the low triangle of the matrix \( A \).
If \( matdescra[1]='U' \), then \( val \) contains elements from the upper triangle of the matrix \( A \).
Refer to values array description in Skyline Storage Scheme for more details.

pntr

Array of length \((m + 1)\) for lower triangle, and \((k + 1)\) for upper triangle.

It contains the indices specifying in the val the positions of the first element in each row (column) of the matrix \(A\). Refer to pointers array description in Skyline Storage Scheme for more details.

\(x\)

Array, size at least \(k\) if \(\text{transa} = 'N'\) or \('n'\) and at least \(m\) otherwise. On entry, the array \(x\) must contain the vector \(x\).

\(beta\)

Specifies the scalar \(beta\).

\(y\)

Array, size at least \(m\) if \(\text{transa} = 'N'\) or \('n'\) and at least \(k\) otherwise. On entry, the array \(y\) must contain the vector \(y\).

Output Parameters

\(y\)

Overwritten by the updated vector \(y\).

mkl_?diasv

Solves a system of linear equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

Syntax

void mkl_sdiasv (const char *transa , const MKL_INT *m , const float *alpha , const char *matdescra , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *x , float *y );
void mkl_ddiasv (const char *transa , const MKL_INT *m , const double *alpha , const char *matdescra , const double *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *x , double *y );
void mkl_cdiasv (const char *transa , const MKL_INT *m , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_INT *x , MKL_Complex8 *y );
void mkl_zdiasv (const char *transa , const MKL_INT *m , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *x , MKL_Complex16 *y );

Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_?_trsv from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?diasv routine solves a system of linear equations with matrix-vector operations for a sparse matrix stored in the diagonal format:

\[ y := \alpha\text{inv}(A)\times \]
or

\[ y := \alpha \cdot \text{inv}(A^T) \cdot x, \]

where:

- \( \alpha \) is scalar,
- \( x \) and \( y \) are vectors,
- \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal,
- \( A^T \) is the transpose of \( A \).

**NOTE**
This routine supports only one-based indexing of the input arrays.

### Input Parameters

- **transa**: Specifies the system of linear equations.
  - If \( \text{transa} = 'N' \) or \( 'n' \), then \( y := \alpha \cdot \text{inv}(A) \cdot x \)
  - If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( y := \alpha \cdot \text{inv}(A^T) \cdot x \)

- **m**: Number of rows of the matrix \( A \).

- **alpha**: Specifies the scalar \( \alpha \).

- **matdescra**: Array of six elements, specifies properties of the matrix used for operation.
  Only first four array elements are used, their possible values are given in Table “Possible Values of the Parameter \( \text{matdescra} \) (\( \text{descra} \))”. Possible combinations of element values of this parameter are given in Table “Possible Combinations of Element Values of the Parameter \( \text{matdescra} \)”.

- **val**: Two-dimensional array of size \( lval \) by \( ndiag \), contains non-zero diagonals of the matrix \( A \). Refer to \( \text{values} \) array description in Diagonal Storage Scheme for more details.

- **lval**: Leading dimension of \( \text{val} \), \( lval \geq m \). Refer to \( lval \) description in Diagonal Storage Scheme for more details.

- **idiag**: Array of length \( ndiag \), contains the distances between main diagonal and each non-zero diagonals in the matrix \( A \).
  - **NOTE**
    All elements of this array must be sorted in increasing order.

Refer to \( \text{distance} \) array description in Diagonal Storage Scheme for more details.

- **ndiag**: Specifies the number of non-zero diagonals of the matrix \( A \).

- **x**: Array, size at least \( m \).
  - On entry, the array \( x \) must contain the vector \( x \). The elements are accessed with unit increment.

- **y**: Array, size at least \( m \).
  - On entry, the array \( y \) must contain the vector \( y \). The elements are accessed with unit increment.
Output Parameters

\( y \)        Contains solution vector \( x \).

\textbf{mkl\_?skysv}  
\textit{Solves a system of linear equations for a sparse matrix in the skyline format with one-based indexing (deprecated).}

\textbf{Syntax}

\begin{verbatim}
void mkl_sskysv (const char *transa, const MKL_INT *m, const float *alpha, const char *matdescra, const float *val, const MKL_INT *pntr, const float *x, float *y);
void mkl_dskysv (const char *transa, const MKL_INT *m, const double *alpha, const char *matdescra, const double *val, const MKL_INT *pntr, const double *x, double *y);
void mkl_cskysv (const char *transa, const MKL_INT *m, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *pntr, const MKL_Complex8 *x, MKL_Complex8 *y);
void mkl_zskysv (const char *transa, const MKL_INT *m, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *pntr, const MKL_Complex16 *x, MKL_Complex16 *y);
\end{verbatim}

\textbf{Include Files}

- \texttt{mkl.h}

\textbf{Description}

This routine is deprecated. Use \texttt{mkl\_sparse\_?\_trsv} from the Intel\textsuperscript{®} MKL Inspector-executor Sparse BLAS interface instead.

The \texttt{mkl\_?skysv} routine solves a system of linear equations with matrix-vector operations for a sparse matrix in the skyline storage format:

\[
y := \alpha \cdot \text{inv}(A) \cdot x
\]

or

\[
y := \alpha \cdot \text{inv}(A^T) \cdot x,
\]

where:

\( \alpha \) is scalar, \( x \) and \( y \) are vectors, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

\textbf{NOTE}

This routine supports only one-based indexing of the input arrays.

\textbf{Input Parameters}

\textit{transa}        Specifies the system of linear equations.

If \texttt{transa} = \texttt{'N'} or \texttt{'n'}, then \( y := \alpha \cdot \text{inv}(A) \cdot x \)
If $transa = 'T'$ or $'t'$ or $'C'$ or $'c'$, then $y := alpha*\text{inv}(A^T) \times x$.

**m**
Number of rows of the matrix $A$.

**alpha**
Specifies the scalar $alpha$.

**matdescra**
Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**NOTE**
General matrices $(matdescra[0]='G')$ is not supported.

**val**
Array containing the set of elements of the matrix $A$ in the skyline profile form.

If $\text{matdescra}[2]='L'$, then $val$ contains elements from the low triangle of the matrix $A$.

If $\text{matdescsa}[2]='U'$, then $val$ contains elements from the upper triangle of the matrix $A$.

Refer to $values$ array description in Skyline Storage Scheme for more details.

**pntr**
Array of length $(m + 1)$ for lower triangle, and $(k + 1)$ for upper triangle.

It contains the indices specifying in the $val$ the positions of the first element in each row (column) of the matrix $A$. Refer to $pointers$ array description in Skyline Storage Scheme for more details.

**x**
Array, size at least $m$.

On entry, the array $x$ must contain the vector $x$. The elements are accessed with unit increment.

**y**
Array, size at least $m$.

On entry, the array $y$ must contain the vector $y$. The elements are accessed with unit increment.

**Output Parameters**

**y**
Contains solution vector $x$.

**mkl_?diamm**

Computes matrix-matrix product of a sparse matrix stored in the diagonal format with one-based indexing (deprecated).
**Syntax**

```c
void mkl_sdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const float *alpha , const char *matdescra , const float *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const float *b , const MKL_INT *ldb , const float *beta , float *c , const MKL_INT *ldc );

void mkl_ddiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const double *alpha , const char *matdescra , const double *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const double *b , const MKL_INT *ldb , const double *beta , double *c , const MKL_INT *ldc );

void mkl_cdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_Complex8 *alpha , const char *matdescra , const MKL_Complex8 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex8 *b , const MKL_INT *ldb , const MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ldc );

void mkl_zdiamm (const char *transa , const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , const MKL_Complex16 *alpha , const char *matdescra , const MKL_Complex16 *val , const MKL_INT *lval , const MKL_INT *idiag , const MKL_INT *ndiag , const MKL_Complex16 *b , const MKL_INT *ldb , const MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ldc );
```

**Description**

This routine is deprecated. Use `mkl_sparse_?_mm` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The **mkl_?diamm** routine performs a matrix-matrix operation defined as

\[
C := \alpha A B + \beta C
\]

or

\[
C := \alpha A^T B + \beta C,
\]

or

\[
C := \alpha A^H B + \beta C,
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( B \) and \( C \) are dense matrices, \( A \) is an \( m \)-by-\( k \) sparse matrix in the diagonal format, \( A^T \) is the transpose of \( A \), and \( A^H \) is the conjugate transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

**Input Parameters**

- **transa**

  Specifies the operation.

  If `transa = 'N'` or `'n'`, then \( C := \alpha A B + \beta C \),
If transa = 'T' or 't', then \( C := \alpha A^T B + \beta C \),

If transa = 'C' or 'c', then \( C := \alpha A^H B + \beta C \).

\( m \)

Number of rows of the matrix \( A \).

\( n \)

Number of columns of the matrix \( C \).

\( k \)

Number of columns of the matrix \( A \).

\( \alpha \)

Specifies the scalar \( \alpha \).

\( \text{matdescra} \)

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter \text{matdescra} (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter \text{matdescra}".

\( \text{val} \)

Two-dimensional array of size \( lval \) by \( ndiag \), contains non-zero diagonals of the matrix \( A \). Refer to values array description in Diagonal Storage Scheme for more details.

\( lval \)

Leading dimension of \( \text{val} \), \( lval \geq m \). Refer to \( lval \) description in Diagonal Storage Scheme for more details.

\( \text{idiag} \)

Array of length \( ndiag \), contains the distances between main diagonal and each non-zero diagonals in the matrix \( A \). Refer to distance array description in Diagonal Storage Scheme for more details.

\( ndiag \)

Specifies the number of non-zero diagonals of the matrix \( A \).

\( \text{b} \)

Array, size \( ldb \times n \).

On entry with \( \text{transa} = 'N' \) or \( 'n' \), the leading \( k \)-by-\( n \) part of the array \( \text{b} \) must contain the matrix \( B \), otherwise the leading \( m \)-by-\( n \) part of the array \( \text{b} \) must contain the matrix \( B \).

\( ldb \)

Specifies the leading dimension of \( \text{b} \) as declared in the calling (sub)program.

\( \beta \)

Specifies the scalar \( \beta \).

\( \text{c} \)

Array, size \( ldc \) by \( n \).

On entry, the leading \( m \)-by-\( n \) part of the array \( \text{c} \) must contain the matrix \( C \), otherwise the leading \( k \)-by-\( n \) part of the array \( \text{c} \) must contain the matrix \( C \).

\( ldc \)

Specifies the leading dimension of \( \text{c} \) as declared in the calling (sub)program.

**Output Parameters**

\( \text{c} \)

Overwritten by the matrix \( (\alpha A B + \beta C), (\alpha A^T B + \beta C), \text{or} (\alpha A^H B + \beta C) \).
**mkl_?skymm**  
*Computes matrix-matrix product of a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).*

**Syntax**

```c
void mkl_sskymm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const float *alpha, const char *matdescra, const float *val, const MKL_INT *pntr, const float *b, const MKL_INT *ldb, const float *beta, float *c, const MKL_INT *ldc);

void mkl_dskymm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const double *alpha, const char *matdescra, const double *val, const MKL_INT *pntr, const double *b, const MKL_INT *ldb, const double *beta, double *c, const MKL_INT *ldc);

void mkl_cskymm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *pntr, const MKL_Complex8 *b, const MKL_INT *ldb, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ldc);

void mkl_zskymm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *pntr, const MKL_Complex16 *b, const MKL_INT *ldb, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ldc);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use `mkl_sparse_?_mm` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_?skymm` routine performs a matrix-matrix operation defined as

```
C := alpha*A*B + beta*C
```

or

```
C := alpha*Aᵀ*B + beta*C,
```

or

```
C := alpha*Aᴴ*B + beta*C,
```

where:

- *alpha* and *beta* are scalars,
- *B* and *C* are dense matrices, *A* is an *m*-by-*k* sparse matrix in the skyline storage format, *Aᵀ* is the transpose of *A*, and *Aᴴ* is the conjugate transpose of *A*.

**NOTE**

This routine supports only one-based indexing of the input arrays.

**Input Parameters**

BLAS and Sparse BLAS Routines
transa

Specifies the operation.
If transa = 'N' or 'n', then $C := \alpha A B + \beta C$,
If transa = 'T' or 't', then $C := \alpha A^\top B + \beta C$,
If transa = 'C' or 'c', then $C := \alpha A^H B + \beta C$.

m

Number of rows of the matrix A.

n

Number of columns of the matrix C.

k

Number of columns of the matrix A.

alpha

Specifies the scalar $\alpha$.

matdescra

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

NOTE

General matrices (matdescra[0]='G') is not supported.

val

Array containing the set of elements of the matrix $A$ in the skyline profile form.
If matdescrsa[2]= 'L', then val contains elements from the low triangle of the matrix $A$.
If matdescrsa[2]= 'U', then val contains elements from the upper triangle of the matrix $A$.
Refer to values array description in Skyline Storage Scheme for more details.

pntr

Array of length $(m + 1)$ for lower triangle, and $(k + 1)$ for upper triangle. It contains the indices specifying the positions of the first element of the matrix $A$ in each row (for the lower triangle) or column (for upper triangle) in the val array such that val[pntr[i] - 1] is the first element in row or column $i + 1$. Refer to pointers array description in Skyline Storage Scheme for more details.

b

Array, size $ldb \times n$.
On entry with transa = 'N' or 'n', the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$, otherwise the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.

ldb

Specifies the leading dimension of $b$ as declared in the calling (sub)program.

beta

Specifies the scalar $\beta$.

c

Array, size $ldc$ by $n$. 
On entry, the leading $m$-by-$n$ part of the array $c$ must contain the matrix $C$, otherwise the leading $k$-by-$n$ part of the array $c$ must contain the matrix $C$.

$\text{ldc}$

Specifies the leading dimension of $c$ as declared in the calling (sub)program.

**Output Parameters**

$c$

Overwritten by the matrix $(\alpha A B + \beta C)$, $(\alpha A^T B + \beta C)$, or $(\alpha A^H B + \beta C)$.

**mkl_?diasm**

Solves a system of linear matrix equations for a sparse matrix in the diagonal format with one-based indexing (deprecated).

**Syntax**

```c
void mkl_sdiasm (const char *transa, const MKL_INT *m, const MKL_INT *n, const float *alpha, const char *matdescra, const float *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const float *b, const MKL_INT *ldb, float *c, const MKL_INT *ldc);
void mkl_ddiasm (const char *transa, const MKL_INT *m, const MKL_INT *n, const double *alpha, const char *matdescra, const double *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const double *b, const MKL_INT *ldb, double *c, const MKL_INT *ldc);
void mkl_cdiasm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex8 *b, const MKL_INT *ldb, MKL_Complex8 *c, const MKL_INT *ldc);
void mkl_zdiasm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *lval, const MKL_INT *idiag, const MKL_INT *ndiag, const MKL_Complex16 *b, const MKL_INT *ldb, MKL_Complex16 *c, const MKL_INT *ldc);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use `mkl_sparse_?_trsm` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_?diasm` routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the diagonal format:

$$C := \alpha \text{inv}(A) B$$

or

$$C := \alpha \text{inv}(A^T) B,$$

where:

$\alpha$ is scalar, $B$ and $C$ are dense matrices, $A$ is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, $A^T$ is the transpose of $A$. 
NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

**transa**
Specifies the system of linear equations.
If \( transa = 'N' \) or \( \text{'}n\text{'} \), then \( C := \alpha \cdot \text{inv}(A) \cdot B \),
If \( transa = 'T' \) or \( \text{'}t\text{'} \) or \( 'C' \) or \( \text{'}c\text{'} \), then \( C := \alpha \cdot \text{inv}(A^T) \cdot B \).

**m**
Number of rows of the matrix \( A \).

**n**
Number of columns of the matrix \( C \).

**alpha**
Specifies the scalar \( \alpha \).

**matdescra**
Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descra)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**val**
Two-dimensional array of size \( lval \) by \( ndiag \), contains non-zero diagonals of the matrix \( A \). Refer to values array description in Diagonal Storage Scheme for more details.

**lval**
Leading dimension of \( val \), \( lval \geq m \). Refer to \( lval \) description in Diagonal Storage Scheme for more details.

**idiag**
Array of length \( ndiag \), contains the distances between main diagonal and each non-zero diagonals in the matrix \( A \).

NOTE
All elements of this array must be sorted in increasing order.

Refer to distance array description in Diagonal Storage Scheme for more details.

**ndiag**
Specifies the number of non-zero diagonals of the matrix \( A \).

**b**
Array, size \( ldb \cdot n \).
On entry the leading \( m \)-by-\( n \) part of the array \( b \) must contain the matrix \( B \).

**ldb**
Specifies the leading dimension of \( b \) as declared in the calling (sub)program.

**ldc**
Specifies the leading dimension of \( c \) as declared in the calling (sub)program.

Output Parameters

**c**
Array, size \( ldc \) by \( n \).
The leading \( m \)-by-\( n \) part of the array \( c \) contains the matrix \( C \).
**mkl_?skysm**

Solves a system of linear matrix equations for a sparse matrix stored using the skyline storage scheme with one-based indexing (deprecated).

**Syntax**

void mkl_sskysm (const char *transa, const MKL_INT *m, const MKL_INT *n, const float *alpha, const char *matdescra, const float *val, const MKL_INT *pntr, const float *b, const MKL_INT *ldb, float *c, const MKL_INT *ldc);

void mkl_dskysm (const char *transa, const MKL_INT *m, const MKL_INT *n, const double *alpha, const char *matdescra, const double *val, const MKL_INT *pntr, const double *b, const MKL_INT *ldb, double *c, const MKL_INT *ldc);

void mkl_cskysm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const char *matdescra, const MKL_Complex8 *val, const MKL_INT *pntr, const MKL_Complex8 *b, const MKL_INT *ldb, MKL_Complex8 *c, const MKL_INT *ldc);

void mkl_zskysm (const char *transa, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const char *matdescra, const MKL_Complex16 *val, const MKL_INT *pntr, const MKL_Complex16 *b, const MKL_INT *ldb, MKL_Complex16 *c, const MKL_INT *ldc);

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use mkl_sparse_?_trsm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?skysm routine solves a system of linear equations with matrix-matrix operations for a sparse matrix in the skyline storage format:

\[ C := \alpha \cdot \text{inv}(A) \cdot B \]

or

\[ C := \alpha \cdot \text{inv}(A^T) \cdot B, \]

where:

- \( \alpha \) is scalar, \( B \) and \( C \) are dense matrices, \( A \) is a sparse upper or lower triangular matrix with unit or non-unit main diagonal, \( A^T \) is the transpose of \( A \).

**NOTE**

This routine supports only one-based indexing of the input arrays.

**Input Parameters**

**transa**

Specifies the system of linear equations.

- If \( \text{transa} = 'N' \) or \( 'n' \), then \( C := \alpha \cdot \text{inv}(A) \cdot B \),
- If \( \text{transa} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( C := \alpha \cdot \text{inv}(A^T) \cdot B \),
Number of rows of the matrix $A$.  

Number of columns of the matrix $C$.  

Specifies the scalar $alpha$.  

Array of six elements, specifies properties of the matrix used for operation. Only first four array elements are used, their possible values are given in Table "Possible Values of the Parameter matdescra (descrA)". Possible combinations of element values of this parameter are given in Table "Possible Combinations of Element Values of the Parameter matdescra".

**NOTE**  
General matrices ($matdescra[0]='G'$) is not supported.

Array containing the set of elements of the matrix $A$ in the skyline profile form.  

If $matdescra[2]='L'$, then $val$ contains elements from the lower triangle of the matrix $A$.  

If $matdescra[2]='U'$, then $val$ contains elements from the upper triangle of the matrix $A$.  

Refer to values array description in Skyline Storage Scheme for more details.

Array of length $(m+1)$ for lower triangle, and $(n+1)$ for upper triangle.  

It contains the indices specifying the positions of the first element of the matrix $A$ in each row (for the lower triangle) or column (for upper triangle) in the $val$ array such that $val[pntr[i]-1]$ is the first element in row or column $i+1$. Refer to pointers array description in Skyline Storage Scheme for more details.

Array, size $ldb* n$.  

On entry the leading $m$-by-$n$ part of the array $b$ must contain the matrix $B$.  

Specifies the leading dimension of $b$ as declared in the calling (sub)program.

Specifies the leading dimension of $c$ as declared in the calling (sub)program.

Output Parameters  

Array, size $ldc$ by $n$.  

The leading $m$-by-$n$ part of the array $c$ contains the matrix $C$.  

$mkl_\text{?dnscsr}$  
*Convert a sparse matrix in uncompressed representation to the CSR format and vice versa (deprecated).*
Syntax

```c
void mkl_ddnscsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *n, double *adns, const MKL_INT *lda, double *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
void mkl_sdncsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *n, float *adns, const MKL_INT *lda, float *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
void mkl_cdnncsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *n, MKL_Complex8 *adns, const MKL_INT *lda, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
void mkl_zdnncsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *n, MKL_Complex16 *adns, const MKL_INT *lda, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *info);
```

Include Files

- mkl.h

Description

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix $A$ between formats: stored as a rectangular array (dense representation) and stored using compressed sparse row (CSR) format (3-array variation).

Input Parameters

job

- **Array**, contains the following conversion parameters:
  - **job[0]**: Conversion type.
    - If job[0]=0, the rectangular matrix $A$ is converted to the CSR format;
    - if job[0]=1, the rectangular matrix $A$ is restored from the CSR format.
  - **job[1]**: Index base for the rectangular matrix $A$.
    - If job[1]=0, zero-based indexing for the rectangular matrix $A$ is used;
    - if job[1]=1, one-based indexing for the rectangular matrix $A$ is used.
  - **job[2]**: Index base for the matrix in CSR format.
    - If job[2]=0, zero-based indexing for the matrix in CSR format is used;
    - if job[2]=1, one-based indexing for the matrix in CSR format is used.
  - **job[3]**: Portion of matrix.
    - If job[3]=0, adns is a lower triangular part of matrix $A$;
    - If job[3]=1, adns is an upper triangular part of matrix $A$;
    - If job[3]=2, adns is a whole matrix $A$.
  - **job[4]**=nzmax: maximum number of the non-zero elements allowed if job[0]=0.
  - **job[5]**: Job indicator for conversion to CSR format.
    - If job[5]=0, only array ia is generated for the output storage.
If $\text{job}[5] > 0$, arrays $\text{acsr}, \text{ia}, \text{ja}$ are generated for the output storage.

$m$
Number of rows of the matrix $A$.

$n$
Number of columns of the matrix $A$.

$\text{adns}$
(input/output)
If the conversion type is from uncompressed to CSR, on input $\text{adns}$ contains an uncompressed (dense) representation of matrix $A$.

$\text{lda}$
Specifies the leading dimension of $\text{adns}$ as declared in the calling (sub)program.
For zero-based indexing of $A$, $\text{lda}$ must be at least $\max(1, n)$.
For one-based indexing of $A$, $\text{lda}$ must be at least $\max(1, m)$.

$\text{acsr}$
(input/output)
If conversion type is from CSR to uncompressed, on input $\text{acsr}$ contains the non-zero elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

$\text{ja}$
(input/output). If conversion type is from CSR to uncompressed, on input for zero-based indexing of $A$ $\text{ja}$ contains the column indices plus one for each non-zero element of the matrix $A$. For one-based indexing of $A$ $\text{ja}$ contains the column indices for each non-zero element of the matrix $A$.
Its length is equal to the length of the array $\text{acsr}$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

$\text{ia}$
(input/output). Array of length $m + 1$.
If conversion type is from CSR to uncompressed, on input for zero-based indexing of $A$ $\text{ia}$ contains indices of elements in the array $\text{acsr}$, such that $\text{ia}[i] - 1$ is the index in the array $\text{acsr}$ of the first non-zero element from the row $i$. For one-based indexing of $A$ $\text{ia}$ contains indices of elements in the array $\text{acsr}$, such that $\text{ia}[i]$ is the index in the array $\text{acsr}$ of the first non-zero element from the row $i$.
The value of $\text{ia}[m] - \text{ia}[0]$ is equal to the number of non-zeros. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

Output Parameters

$\text{adns}$
If conversion type is from CSR to uncompressed, on output $\text{adns}$ contains the uncompressed (dense) representation of matrix $A$.

$\text{acsr}, \text{ja}, \text{ia}$
If conversion type is from uncompressed to CSR, on output $\text{acsr}, \text{ja}, \text{ia}$ contain the compressed sparse row (CSR) format (3-array variation) of matrix $A$ (see Sparse Matrix Storage Formats for a description of the storage format).

$\text{info}$
Integer info indicator only for restoring the matrix $A$ from the CSR format.
If $\text{info}=0$, the execution is successful.
If \( \text{info}=i \), the routine is interrupted processing the \( i \)-th row because there is no space in the arrays \( acsr \) and \( ja \) according to the value \( nzmax \).

**mkl_?csrcoo**

Converting a sparse matrix in the CSR format to the coordinate format and vice versa (deprecated).

**Syntax**

```c
void mkl_scsrcoo (const MKL_INT *job, const MKL_INT *n, float *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *nnz, float *acoo, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
void mkl_dcsrcoo (const MKL_INT *job, const MKL_INT *n, double *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *nnz, double *acoo, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
void mkl_ccsrcoo (const MKL_INT *job, const MKL_INT *n, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *nnz, MKL_Complex8 *acoo, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
void mkl_zcsrcoo (const MKL_INT *job, const MKL_INT *n, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_INT *nnz, MKL_Complex16 *acoo, MKL_INT *rowind, MKL_INT *colind, MKL_INT *info);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix \( A \) stored in the compressed sparse row (CSR) format (3-array variation) to coordinate format and vice versa.

**Input Parameters**

- **job**

  Array, contains the following conversion parameters:

  - **job[0]**
    - If \( \text{job[0]}=0 \), the matrix in the CSR format is converted to the coordinate format;
    - if \( \text{job[0]}=1 \), the matrix in the coordinate format is converted to the CSR format.
    - if \( \text{job[0]}=2 \), the matrix in the coordinate format is converted to the CSR format, and the column indices in CSR representation are sorted in the increasing order within each row.

  - **job[1]**
    - If \( \text{job[1]}=0 \), zero-based indexing for the matrix in CSR format is used;
    - if \( \text{job[1]}=1 \), one-based indexing for the matrix in CSR format is used.

  - **job[2]**
If \( \text{job}[2]=0 \), zero-based indexing for the matrix in coordinate format is used;
if \( \text{job}[2]=1 \), one-based indexing for the matrix in coordinate format is used.

\( \text{job}[4]=\text{nzmax} \) - maximum number of the non-zero elements allowed if \( \text{job}[0]=0 \).

\( \text{job}[5] \) - job indicator.

For conversion to the coordinate format:
If \( \text{job}[5]=1 \), only array \( \text{rowind} \) is filled in for the output storage.
If \( \text{job}[5]=2 \), arrays \( \text{rowind}, \text{colind} \) are filled in for the output storage.
If \( \text{job}[5]=3 \), all arrays \( \text{rowind}, \text{colind}, \text{acoo} \) are filled in for the output storage.

For conversion to the CSR format:
If \( \text{job}[5]=0 \), all arrays \( \text{acsr}, \text{ja}, \text{ia} \) are filled in for the output storage.
If \( \text{job}[5]=1 \), only array \( \text{ia} \) is filled in for the output storage.
If \( \text{job}[5]=2 \), then it is assumed that the routine already has been called with the \( \text{job}[5]=1 \), and the user allocated the required space for storing the output arrays \( \text{acsr} \) and \( \text{ja} \).

\( n \) - Dimension of the matrix \( A \).

\( \text{nnz} \) - Specifies the number of non-zero elements of the matrix \( A \) for \( \text{job}[0] \neq 0 \).
Refer to \( \text{nnz} \) description in Coordinate Format for more details.

\( \text{acsr} \) - (input/output) Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to values array description in Sparse Matrix Storage Formats for more details.

\( \text{ja} \) - (input/output). For \( \text{job}[1] = 1 \) (one-based indexing for the matrix in CSR format), array containing the column indices plus one for each non-zero element of the matrix \( A \).

For \( \text{job}[1] = 0 \) (zero-based indexing for the matrix in CSR format), array containing the column indices for each non-zero element of the matrix \( A \). Its length is equal to the length of the array \( \text{acsr} \). Refer to columns array description in Sparse Matrix Storage Formats for more details.

\( \text{ia} \) - (input/output). Array of length \( n + 1 \), containing indices of elements in the array \( \text{acsr} \), such that \( \text{ia}[i] - \text{ia}[0] \) is the index in the array \( \text{acsr} \) of the first non-zero element from the row \( i \). The value of the last element \( \text{ia}[n] - \text{ia}[0] \) is equal to the number of non-zeros plus one. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

\( \text{acoo} \) - (input/output)
Array containing non-zero elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

*rowind* (input/output). Array of length $nnz$, contains the row indices for each non-zero element of the matrix $A$. Refer to rows array description in Coordinate Format for more details.

*colind* (input/output). Array of length $nnz$, contains the column indices for each non-zero element of the matrix $A$. Refer to columns array description in Coordinate Format for more details.

**Output Parameters**

*nnz* Returns the number of converted elements of the matrix $A$ for $job[0]=0$.

*info* Integer info indicator only for converting the matrix $A$ from the CSR format.

If $info=0$, the execution is successful.

If $info=1$, the routine is interrupted because there is no space in the arrays $aco$, $rowind$, $colind$ according to the value $nzmax$.

**mkl_?csrbsr**

*Converts a square sparse matrix in the CSR format to the BSR format and vice versa (deprecated).*

**Syntax**

```c
void mkl_scsrbsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *mblk, const MKL_INT *ldabsr, float *acsr, MKL_INT *ja, MKL_INT *ia, float *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT *info);
void mkl_dcsrbsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *mblk, const MKL_INT *ldabsr, double *acsr, MKL_INT *ja, MKL_INT *ia, double *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT *info);
void mkl_ccsrbsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *mblk, const MKL_INT *ldabsr, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT *info);
void mkl_zcsrbsr (const MKL_INT *job, const MKL_INT *m, const MKL_INT *mblk, const MKL_INT *ldabsr, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *absr, MKL_INT *jab, MKL_INT *iab, MKL_INT *info);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix $A$ stored in the compressed sparse row (CSR) format (3-array variation) to the block sparse row (BSR) format and vice versa.
Input Parameters

**job**

Array, contains the following conversion parameters:

- **job[0]**
  - If job[0]==0, the matrix in the CSR format is converted to the BSR format;
  - if job[0]==1, the matrix in the BSR format is converted to the CSR format.

- **job[1]**
  - If job[1]==0, zero-based indexing for the matrix in CSR format is used;
  - if job[1]==1, one-based indexing for the matrix in CSR format is used.

- **job[2]**
  - If job[2]==0, zero-based indexing for the matrix in the BSR format is used;
  - if job[2]==1, one-based indexing for the matrix in the BSR format is used.

- **job[3]** is only used for conversion to CSR format. By default, the converter saves the blocks without checking whether an element is zero or not. If job[3]==1, then the converter only saves non-zero elements in blocks.

- **job[5]** - job indicator.
  - For conversion to the BSR format:
    - If job[5]==0, only arrays jab, iab are generated for the output storage.
    - If job[5]>0, all output arrays absr, jab, and iab are filled in for the output storage.
    - If job[5]==-1, iab[m] - iab[0] returns the number of non-zero blocks.
  - For conversion to the CSR format:
    - If job[5]==0, only arrays ja, ia are generated for the output storage.

**m**

Actual row dimension of the matrix A for convert to the BSR format; block row dimension of the matrix A for convert to the CSR format.

**mblk**

Size of the block in the matrix A.

**ldabsr**

Leading dimension of the array absr as declared in the calling program. ldabsr must be greater than or equal to mblk*mblk.

**acsr**

Array containing non-zero elements of the matrix A. Its length is equal to the number of non-zero elements in the matrix A. Refer to values array description in Sparse Matrix Storage Formats for more details.

**ja**

Array containing the column indices for each non-zero element of the matrix A.

Its length is equal to the length of the array acsr. Refer to columns array description in Sparse Matrix Storage Formats for more details.
(input/output). Array of length \( m + 1 \), containing indices of elements in the array \( \text{acsr} \), such that \( \text{ia}[i] - \text{iab}[0] \) is the index in the array \( \text{acsr} \) of the first non-zero element from the row \( i \). The value of \( \text{ia}[m] - \text{iab}[0] \) is equal to the number of non-zeros. Refer to \( \text{rowIndex} \) array description in Sparse Matrix Storage Formats for more details.

\( \text{absr} \)

(input/output)
Array containing elements of non-zero blocks of the matrix \( A \). Its length is equal to the number of non-zero blocks in the matrix \( A \) multiplied by \( mblk \times mblk \). Refer to \( \text{values} \) array description in BSR Format for more details.

\( \text{jab} \)

(input/output). Array containing the column indices for each non-zero block of the matrix \( A \).
Its length is equal to the number of non-zero blocks of the matrix \( A \). Refer to \( \text{columns} \) array description in BSR Format for more details.

\( \text{iab} \)

(input/output). Array of length \( (m + 1) \), containing indices of blocks in the array \( \text{absr} \), such that \( \text{iab}[i] - \text{iab}[0] \) is the index in the array \( \text{absr} \) of the first non-zero element from the \( i \)-th row. The value of \( \text{iab}[m] \) is equal to the number of non-zero blocks. Refer to \( \text{rowIndex} \) array description in BSR Format for more details.

Output Parameters

\( \text{info} \)

Integer info indicator only for converting the matrix \( A \) from the CSR format.
If \( \text{info}=0 \), the execution is successful.
If \( \text{info}=1 \), it means that \( mblk \) is equal to 0.
If \( \text{info}=2 \), it means that \( \text{ldabsr} \) is less than \( mblk \times mblk \) and there is no space for all blocks.

\textbf{mkl\_?csrsc}

\textit{Converts a square sparse matrix in the CSR format to the CSC format and vice versa (deprecated).}

\textbf{Syntax}

void mkl_dcsrsc (const MKL_INT *job, const MKL_INT *n, double *acsr, MKL_INT *ja, MKL_INT *ia, double *acsc, MKL_INT *ja1, MKL_INT *ia1, MKL_INT *info);
void mkl_scsrsc (const MKL_INT *job, const MKL_INT *n, float *acsr, MKL_INT *ja, MKL_INT *ia, float *acsc, MKL_INT *ja1, MKL_INT *ia1, MKL_INT *info);
void mkl_ccsrsc (const MKL_INT *job, const MKL_INT *n, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *acsc, MKL_INT *ja1, MKL_INT *ia1, MKL_INT *info);
void mkl_zcsrsc (const MKL_INT *job, const MKL_INT *n, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *acsc, MKL_INT *ja1, MKL_INT *ia1, MKL_INT *info);

\textbf{Include Files}

- mkl.h
Description

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

This routine converts a square sparse matrix $A$ stored in the compressed sparse row (CSR) format (3-array variation) to the compressed sparse column (CSC) format and vice versa.

**Input Parameters**

- **job**
  
  Array, contains the following conversion parameters:
  
  - *job[0]*
    
    If *job[0]=0*, the matrix in the CSR format is converted to the CSC format;
    
    if *job[0]=1*, the matrix in the CSC format is converted to the CSR format.
  
  - *job[1]*
    
    If *job[1]=0*, zero-based indexing for the matrix in CSR format is used;
    
    if *job[1]=1*, one-based indexing for the matrix in CSR format is used.
  
  - *job[2]*
    
    If *job[2]=0*, zero-based indexing for the matrix in the CSC format is used;
    
    if *job[2]=1*, one-based indexing for the matrix in the CSC format is used.
  
  - *job[5]* - job indicator.
    
    For conversion to the CSC format:
    
    - If *job[5]=0*, only arrays $ja1$, $ia1$ are filled in for the output storage.
    
    - If *job[5]≠0*, all output arrays $acsc$, $ja1$, and $ia1$ are filled in for the output storage.

    For conversion to the CSR format:
    
    - If *job[5]=0*, only arrays $ja$, $ia$ are filled in for the output storage.
    
    - If *job[5]≠0*, all output arrays $acsr$, $ja$, and $ia$ are filled in for the output storage.

- **m**
  
  Dimension of the square matrix $A$.

- **acsr**
  
  (input/output)
  
  Array containing non-zero elements of the square matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

- **ja**
  
  (input/output). Array containing the column indices for each non-zero element of the matrix $A$.
  
  Its length is equal to the length of the array $acsr$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

- **ia**
  
  (input/output). Array of length $m + 1$, containing indices of elements in the array $acsr$, such that $ia[i] - ia[0]$ is the index in the array $acsr$ of the first non-zero element from the row $i$. The value of $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.
The input/output parameter `acsc` is an array containing non-zero elements of the square matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to the values array description in Sparse Matrix Storage Formats for more details.

The input/output parameter `jal` is an array containing the row indices for each non-zero element of the matrix \( A \). Its length is equal to the length of the array `acsc`. Refer to the columns array description in Sparse Matrix Storage Formats for more details.

The input/output parameter `ial` is an array of length \( m + 1 \), containing indices of elements in the array `acsc`, such that \( ial[i] - ial[0] \) is the index in the array `acsc` of the first non-zero element from the column \( i \). The value of \( ial[m] - ial[0] \) is equal to the number of non-zeros. Refer to the rowIndex array description in Sparse Matrix Storage Formats for more details.

**Output Parameters**

The parameter `info` is not used now.

**mkl_?csrdia**

*Converts a sparse matrix in the CSR format to the diagonal format and vice versa (deprecated).*

**Syntax**

```c
void mkl_dcsrdia (const MKL_INT *job, const MKL_INT *n, double *acsr, MKL_INT *ja, MKL_INT *ia, double *adia, const MKL_INT *ndiag, MKL_INT *distance, MKL_INT *idimag, double *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
void mkl_scsrdia (const MKL_INT *job, const MKL_INT *n, float *acsr, MKL_INT *ja, MKL_INT *ia, float *adia, const MKL_INT *ndiag, MKL_INT *distance, MKL_INT *idimag, float *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
void mkl_ccsrdia (const MKL_INT *job, const MKL_INT *n, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *adia, const MKL_INT *ndiag, MKL_INT *distance, MKL_INT *idimag, MKL_Complex8 *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
void mkl_zcsrdia (const MKL_INT *job, const MKL_INT *n, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *adia, const MKL_INT *ndiag, MKL_INT *distance, MKL_INT *idimag, MKL_Complex16 *acsr_rem, MKL_INT *ja_rem, MKL_INT *ia_rem, MKL_INT *info);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

This routine converts a sparse matrix \( A \) stored in the compressed sparse row (CSR) format (3-array variation) to the diagonal format and vice versa.
Input Parameters

job

Array, contains the following conversion parameters:

job

job[0]
If \(job[0]=0\), the matrix in the CSR format is converted to the diagonal format;
if \(job[0]=1\), the matrix in the diagonal format is converted to the CSR format.

job[1]
If \(job[1]=0\), zero-based indexing for the matrix in CSR format is used;
if \(job[1]=1\), one-based indexing for the matrix in CSR format is used.

job[2]
If \(job[2]=0\), zero-based indexing for the matrix in the diagonal format is used;
if \(job[2]=1\), one-based indexing for the matrix in the diagonal format is used.

For conversion to the diagonal format:

If \(job[5]=0\), diagonals are not selected internally, and \(acsr\_rem, ja\_rem, ia\_rem\) are not filled in for the output storage.
If \(job[5]=1\), diagonals are not selected internally, and \(acsr\_rem, ja\_rem, ia\_rem\) are filled in for the output storage.
If \(job[5]=10\), diagonals are selected internally, and \(acsr\_rem, ja\_rem, ia\_rem\) are not filled in for the output storage.
If \(job[5]=11\), diagonals are selected internally, and \(csr\_rem, ja\_rem, ia\_rem\) are filled in for the output storage.

For conversion to the CSR format:

If \(job[5]=0\), each entry in the array \(adia\) is checked whether it is zero.
Zero entries are not included in the array \(acsr\).
If \(job[5] \neq 0\), each entry in the array \(adia\) is not checked whether it is zero.

m

Dimension of the matrix \(A\).

acsr

(input/output)
Array containing non-zero elements of the matrix \(A\). Its length is equal to the number of non-zero elements in the matrix \(A\). Refer to values array description in Sparse Matrix Storage Formats for more details.

ja

(input/output). Array containing the column indices for each non-zero element of the matrix \(A\).
Its length is equal to the length of the array \(acsr\). Refer to columns array description in Sparse Matrix Storage Formats for more details.
ia (input/output). Array of length \( m + 1 \), containing indices of elements in the array \( acsr \), such that \( ia[i] - ia[0] \) is the index in the array \( acsr \) of the first non-zero element from the row \( i \). The value of \( ia[m] - ia[0] \) is equal to the number of non-zeros. Refer to \( rowIndex \) array description in Sparse Matrix Storage Formats for more details.

adia (input/output)
Array of size \( (ndiag*idiag) \) containing diagonals of the matrix \( A \).

The key point of the storage is that each element in the array \( adia \) retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom.

ndiag
Specifies the leading dimension of the array \( adia \) as declared in the calling (sub)program, must be at least \( \max(1, m) \).

distance
Array of length \( idia \), containing the distances between the main diagonal and each non-zero diagonal to be extracted. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

idiag
Number of diagonals to be extracted. For conversion to diagonal format on return this parameter may be modified.

acsr\_rem, ja\_rem, ia\_rem
Remainder of the matrix in the CSR format if it is needed for conversion to the diagonal format.

Output Parameters

info
This parameter is not used now.

**mkl\_?csrsky**
Converts a sparse matrix in CSR format to the skyline format and vice versa (deprecated).

**Syntax**

```c
void mkl_dcsrsky (const MKL_INT *job, const MKL_INT *m, double *acsr, MKL_INT *ja, MKL_INT *ia, double *asky, MKL_INT *pointers, MKL_INT *info);
void mkl_scsrsky (const MKL_INT *job, const MKL_INT *m, float *acsr, MKL_INT *ja, MKL_INT *ia, float *asky, MKL_INT *pointers, MKL_INT *info);
void mkl_ccsrsky (const MKL_INT *job, const MKL_INT *m, MKL_Complex8 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *asky, MKL_INT *pointers, MKL_INT *info);
void mkl_zcsrsky (const MKL_INT *job, const MKL_INT *m, MKL_Complex16 *acsr, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *asky, MKL_INT *pointers, MKL_INT *info);
```

**Include Files**

- mkl.h

**Description**

This routine is deprecated. Use the matrix manipulation routines from the Intel® MKL Inspector-executor Sparse BLAS interface instead.
This routine converts a sparse matrix \( A \) stored in the compressed sparse row (CSR) format (3-array variation) to the skyline format and vice versa.

**Input Parameters**

\( job \)

Array, contains the following conversion parameters:

\( job[0] \)

If \( job[0]=0 \), the matrix in the CSR format is converted to the skyline format;

if \( job[0]=1 \), the matrix in the skyline format is converted to the CSR format.

\( job[1] \)

If \( job[1]=0 \), zero-based indexing for the matrix in CSR format is used;

if \( job[1]=1 \), one-based indexing for the matrix in CSR format is used.

\( job[2] \)

If \( job[2]=0 \), zero-based indexing for the matrix in the skyline format is used;

if \( job[2]=1 \), one-based indexing for the matrix in the skyline format is used.

\( job[3] \)

For conversion to the skyline format:

If \( job[3]=0 \), the upper part of the matrix \( A \) in the CSR format is converted.

If \( job[3]=1 \), the lower part of the matrix \( A \) in the CSR format is converted.

For conversion to the CSR format:

If \( job[3]=0 \), the matrix is converted to the upper part of the matrix \( A \) in the CSR format.

If \( job[3]=1 \), the matrix is converted to the lower part of the matrix \( A \) in the CSR format.

\( job[4] \)

\( job[4]=nzmax \) - maximum number of the non-zero elements of the matrix \( A \) if \( job[0]=0 \).


Only for conversion to the skyline format:

If \( job[5]=0 \), only arrays \( pointers \) is filled in for the output storage.

If \( job[5]=1 \), all output arrays \( asky \) and \( pointers \) are filled in for the output storage.

\( m \)

Dimension of the matrix \( A \).

\( acsr \)

(input/output)

Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to \( values \) array description in *Sparse Matrix Storage Formats* for more details.
ja

(input/output). Array containing the column indices for each non-zero element of the matrix $A$.

Its length is equal to the length of the array $acsr$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

ia

(input/output). Array of length $m + 1$, containing indices of elements in the array $acsr$, such that $ia[i] - ia[0]$ is the index in the array $acsr$ of the first non-zero element from the row $i$. The value of $ia[m] - ia[0]$ is equal to the number of non-zeros. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

asky

(input/output)

Array, for a lower triangular part of $A$ it contains the set of elements from each row starting from the first none-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets. Refer to values array description in Skyline Storage Format for more details.

pointers

(input/output).

Array with dimension $(m+1)$, where $m$ is number of rows for lower triangle (columns for upper triangle), $pointers[i-1] - pointers[0]$ gives the index of element in the array $asky$ that is first non-zero element in row (column) . The value of $pointers[m]$ is set to $nnz + pointers[0]$, where $nnz$ is the number of elements in the array $asky$. Refer to pointers array description in Skyline Storage Format for more details.

**Output Parameters**

**info**

Integer info indicator only for converting the matrix $A$ from the CSR format.

If $info=0$, the execution is successful.

If $info=1$, the routine is interrupted because there is no space in the array $asky$ according to the value $nzmax$.

**mkl_?csradd**

*Computes the sum of two matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).*

**Syntax**

```c
void mkl_dcsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , double *a , MKL_INT *ja , MKL_INT *ia , const
double *beta , double *b , MKL_INT *jb , MKL_INT *ib , double *c , MKL_INT *jc , MKL_INT
*ic , const MKL_INT *nzmax , MKL_INT *info );

void mkl_scsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , float *a , MKL_INT *ja , MKL_INT *ia , const
float *beta , float *b , MKL_INT *jb , MKL_INT *ib , float *c , MKL_INT *jc , MKL_INT
*ic , const MKL_INT *nzmax , MKL_INT *info );
```
void mkl_ccsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *ia ,
const MKL_Complex8 *beta , MKL_Complex8 *b , MKL_INT *jb , MKL_INT *ib , MKL_Complex8
*c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );

void mkl_zcsradd (const char *trans , const MKL_INT *request , const MKL_INT *sort ,
const MKL_INT *m , const MKL_INT *n , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *ia ,
const MKL_Complex16 *beta , MKL_Complex16 *b , MKL_INT *jb , MKL_INT *ib ,
MKL_Complex16 *c , MKL_INT *jc , MKL_INT *ic , const MKL_INT *nzmax , MKL_INT *info );

Include Files
• mkl.h

Description
This routine is deprecated. Use mkl_sparse_?_add from the Intel® MKL Inspector-executor Sparse BLAS
interface instead.

The mkl_?csradd routine performs a matrix-matrix operation defined as

\[ C := A + \beta \cdot \text{op}(B) \]

where:

- \( A, B, C \) are the sparse matrices in the CSR format (3-array variation).
- \( \text{op}(B) \) is one of \( \text{op}(B) = B, B^T, \) or \( B^H \)
- \( \beta \) is a scalar.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices \( A \) and \( B \) are arranged in the increasing order for each row. If not, use the parameter sort (see below) to reorder column indices and the corresponding elements of the input matrices.

NOTE
This routine supports only one-based indexing of the input arrays.

Input Parameters

- trans
  - Specifies the operation.
  - If \( \text{trans} = 'N' \) or \( 'n' \), then \( C := A + \beta \cdot B \)
  - If \( \text{trans} = 'T' \) or \( 't' \), then \( C := A + \beta \cdot B^T \)
  - If \( \text{trans} = 'C' \) or \( 'c' \), then \( C := A + \beta \cdot B^H \).

- request
  - If request=0, the routine performs addition. The memory for the output arrays ic, jc, c must be allocated beforehand.
  - If request=1, the routine only computes the values of the array ic of length \( m + 1 \). The memory for the ic array must be allocated beforehand. On exit the value \( ic[m] - 1 \) is the actual number of the elements in the arrays c and jc.
  - If request=2, after the routine is called previously with the parameter request=1 and after the output arrays jc and c are allocated in the calling program with length at least \( ic[m] - 1 \), the routine performs addition.
sort

Specifies the type of reordering. If this parameter is not set (default), the routine does not perform reordering.

If sort=1, the routine arranges the column indices $ja$ for each row in the increasing order and reorders the corresponding values of the matrix $A$ in the array $a$.

If sort=2, the routine arranges the column indices $jb$ for each row in the increasing order and reorders the corresponding values of the matrix $B$ in the array $b$.

If sort=3, the routine performs reordering for both input matrices $A$ and $B$.

$m$

Number of rows of the matrix $A$.

$n$

Number of columns of the matrix $A$.

$a$

Array containing non-zero elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

$ja$

Array containing the column indices plus one for each non-zero element of the matrix $A$. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array $a$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

$ia$

Array of length $m + 1$, containing indices of elements in the array $a$, such that $ia[i] - ia[0]$ is the index in the array $a$ of the first non-zero element from the row $i$. The value of the last element $ia[m]$ is equal to the number of non-zero elements of the matrix $A$ plus one. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

$beta$

Specifies the scalar $beta$.

$b$

Array containing non-zero elements of the matrix $B$. Its length is equal to the number of non-zero elements in the matrix $B$. Refer to values array description in Sparse Matrix Storage Formats for more details.

$jb$

Array containing the column indices plus one for each non-zero element of the matrix $B$. For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array $b$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

$ib$

Array of length $m + 1$ when $trans = 'N'$ or $'n'$, or $n + 1$ otherwise.

This array contains indices of elements in the array $b$, such that $ib[i] - ib[0]$ is the index in the array $b$ of the first non-zero element from the row $i$. The value of the last element $ib[m]$ or $ib[n]$ is equal to the number of non-zero elements of the matrix $B$ plus one. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

$nzmax$

The length of the arrays $c$ and $jc$. 
This parameter is used only if \texttt{request}=0. The routine stops calculation if the number of elements in the result matrix $C$ exceeds the specified value of $\texttt{nzmax}$.

**Output Parameters**

$c$

Array containing non-zero elements of the result matrix $C$. Its length is equal to the number of non-zero elements in the matrix $C$. Refer to \textit{values} array description in \textit{Sparse Matrix Storage Formats} for more details.

$jc$

Array containing the column indices plus one for each non-zero element of the matrix $C$.

The length of this array is equal to the length of the array $c$. Refer to \textit{columns} array description in \textit{Sparse Matrix Storage Formats} for more details.

$ic$

Array of length $m + 1$, containing indices of elements in the array $c$, such that $ic[i] - ic[0]$ is the index in the array $c$ of the first non-zero element from the row $i$. The value of the last element $ic[m]$ is equal to the number of non-zero elements of the matrix $C$ plus one. Refer to \textit{rowIndex} array description in \textit{Sparse Matrix Storage Formats} for more details.

$\texttt{info}$

If $\texttt{info}$=0, the execution is successful.

If $\texttt{info}$=I>0, the routine stops calculation in the I-th row of the matrix $C$ because number of elements in $C$ exceeds $\texttt{nzmax}$.

If $\texttt{info}$=-1, the routine calculates only the size of the arrays $c$ and $jc$ and returns this value plus 1 as the last element of the array $ic$.

\texttt{mkl_csrmultcsr}

\textit{Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing (deprecated).}

**Syntax**

```c
void mkl_dcsrmultcsr (const char *trans, const MKL_INT *request, const MKL_INT *sort, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, double *a, MKL_INT *ja, MKL_INT *ia, double *b, MKL_INT *jb, MKL_INT *ib, double *c, MKL_INT *jc, MKL_INT *ic, const MKL_INT *nzmax, MKL_INT *info);

void mkl_scsrmultcsr (const char *trans, const MKL_INT *request, const MKL_INT *sort, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, float *a, MKL_INT *ja, MKL_INT *ia, float *b, MKL_INT *jb, MKL_INT *ib, float *c, MKL_INT *jc, MKL_INT *ic, const MKL_INT *nzmax, MKL_INT *info);

void mkl_ccsrmultcsr (const char *trans, const MKL_INT *request, const MKL_INT *sort, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex8 *c, MKL_INT *jc, MKL_INT *ic, const MKL_INT *nzmax, MKL_INT *info);

void mkl_zcsrmultcsr (const char *trans, const MKL_INT *request, const MKL_INT *sort, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex16 *c, MKL_INT *jc, MKL_INT *ic, const MKL_INT *nzmax, MKL_INT *info);
```
Include Files

• mkl.h

Description

This routine is deprecated. Use mkl_sparse_spmm from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The mkl_?csrmultcsr routine performs a matrix-matrix operation defined as

\[ C := \text{op}(A) \times B \]

where:

- \( A, B, C \) are the sparse matrices in the CSR format (3-array variation);
- \( \text{op}(A) \) is one of \( \text{op}(A) = A, \) or \( \text{op}(A) = A^T, \) or \( \text{op}(A) = A^H. \)

You can use the parameter sort to perform or not perform reordering of non-zero entries in input and output sparse matrices. The purpose of reordering is to rearrange non-zero entries in compressed sparse row matrix so that column indices in compressed sparse representation are sorted in the increasing order for each row.

The following table shows correspondence between the value of the parameter sort and the type of reordering performed by this routine for each sparse matrix involved:

<table>
<thead>
<tr>
<th>Value of the parameter sort</th>
<th>Reordering of ( A ) (arrays ( a, ja, ia ))</th>
<th>Reordering of ( B ) (arrays ( b, ja, ib ))</th>
<th>Reordering of ( C ) (arrays ( c, jc, ic ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>arbitrary value not equal to 1, 2, ..., 7</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

**NOTE**

This routine supports only one-based indexing of the input arrays.

Input Parameters

\[ \text{trans} \]

Specifies the operation.

- If \( \text{trans} = 'N' \) or \( 'n' \), then \( C := A \times B \)
- If \( \text{trans} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), then \( C := A^T \times B. \)

\[ \text{request} \]

If \( \text{request}=0 \), the routine performs multiplication, the memory for the output arrays \( ic, jc, c \) must be allocated beforehand.

If \( \text{request}=1 \), the routine computes only values of the array \( ic \) of length \( m + 1 \), the memory for this array must be allocated beforehand. On exit the value \( ic[m] - 1 \) is the actual number of the elements in the arrays \( c \) and \( jc \).
If \( \text{request}=2 \), the routine has been called previously with the parameter \( \text{request}=1 \), the output arrays \( jc \) and \( c \) are allocated in the calling program and they are of the length \( ic[m] - 1 \) at least.

\textit{sort}

Specifies whether the routine performs reordering of non-zeros entries in input and/or output sparse matrices (see table above).

\( m \)

Number of rows of the matrix \( A \).

\( n \)

Number of columns of the matrix \( A \).

\( k \)

Number of columns of the matrix \( B \).

\( a \)

Array containing non-zero elements of the matrix \( A \). Its length is equal to the number of non-zero elements in the matrix \( A \). Refer to \textit{values} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( ja \)

Array containing the column indices plus one for each non-zero element of the matrix \( A \). For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array \( a \). Refer to \textit{columns} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( ia \)

Array of length \( m + 1 \).

This array contains indices of elements in the array \( a \), such that \( ia[i] - ia[0] \) is the index in the array \( a \) of the first non-zero element from the row \( i \). The value of the last element \( ia[m] \) is equal to the number of non-zero elements of the matrix \( A \) plus one. Refer to \textit{rowIndex} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( b \)

Array containing non-zero elements of the matrix \( B \). Its length is equal to the number of non-zero elements in the matrix \( B \). Refer to \textit{values} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( jb \)

Array containing the column indices plus one for each non-zero element of the matrix \( B \). For each row the column indices must be arranged in the increasing order.

The length of this array is equal to the length of the array \( b \). Refer to \textit{columns} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( ib \)

Array of length \( n + 1 \) when \( \text{trans} = 'N' \) or '\n', or \( m + 1 \) otherwise.

This array contains indices of elements in the array \( b \), such that \( ib[i] - ib[0] \) is the index in the array \( b \) of the first non-zero element from the row \( i \). The value of the last element \( ib[n] \) or \( ib[m] \) is equal to the number of non-zero elements of the matrix \( B \) plus one. Refer to \textit{rowIndex} array description in \textit{Sparse Matrix Storage Formats} for more details.

\( nzmax \)

The length of the arrays \( c \) and \( jc \).

This parameter is used only if \( \text{request}=0 \). The routine stops calculation if the number of elements in the result matrix \( C \) exceeds the specified value of \( nzmax \).
**Output Parameters**

- **c**: Array containing non-zero elements of the result matrix C. Its length is equal to the number of non-zero elements in the matrix C. Refer to values array description in Sparse Matrix Storage Formats for more details.

- **jc**: Array containing the column indices plus one for each non-zero element of the matrix C. The length of this array is equal to the length of the array c. Refer to columns array description in Sparse Matrix Storage Formats for more details.

- **ic**: Array of length \( m + 1 \) when \( \text{trans} = 'N' \) or \( 'n' \), or \( n + 1 \) otherwise. This array contains indices of elements in the array c, such that \( \text{ic}[i] - \text{ic}[0] \) is the index in the array c of the first non-zero element from the row \( i \). The value of the last element \( \text{ic}[m] \) or \( \text{ic}[n] \) is equal to the number of non-zero elements of the matrix C plus one. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

- **info**: If \( \text{info}=0 \), the execution is successful. If \( \text{info}=I>0 \), the routine stops calculation in the \( I \)-th row of the matrix C because number of elements in C exceeds \( \text{nzmax} \). If \( \text{info}=-1 \), the routine calculates only the size of the arrays c and jc and returns this value plus 1 as the last element of the array ic.

**mkl_?csrmultd**

Computes product of two sparse matrices stored in the CSR format (3-array variation) with one-based indexing. The result is stored in the dense matrix (deprecated).

**Syntax**

```c
void mkl_dcsrmultd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, double *a, MKL_INT *ja, MKL_INT *ia, double *b, MKL_INT *jb, MKL_INT *ib, double *c, MKL_INT *ldc);
void mkl_scsrmultd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, float *a, MKL_INT *ja, MKL_INT *ia, float *b, MKL_INT *jb, MKL_INT *ib, float *c, MKL_INT *ldc);
void mkl_ccsrmultd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex8 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex8 *c, MKL_INT *ldc);
void mkl_zcsrmultd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *ia, MKL_Complex16 *b, MKL_INT *jb, MKL_INT *ib, MKL_Complex16 *c, MKL_INT *ldc);
```

**Include Files**

- mkl.h

**Description**
This routine is deprecated. Use `mkl_sparse_?_spmm` from the Intel® MKL Inspector-executor Sparse BLAS interface instead.

The `mkl_?csrmultd` routine performs a matrix-matrix operation defined as

$$ C := \text{op}(A) * B $$

where:

- $A$, $B$ are the sparse matrices in the CSR format (3-array variation), $C$ is dense matrix;
- $\text{op}(A)$ is one of $\text{op}(A) = A$, or $\text{op}(A) = A^T$, or $\text{op}(A) = A^H$.

The routine works correctly if and only if the column indices in sparse matrix representations of matrices $A$ and $B$ are arranged in the increasing order for each row.

**NOTE**
This routine supports only one-based indexing of the input arrays.

**Input Parameters**

- $trans$ Specifies the operation.
  - If $trans = \text{'N'}$ or $\text{'n'}$, then $C := A * B$
  - If $trans = \text{'T'}$ or $\text{'t'}$ or $\text{'C'}$ or $\text{'c'}$, then $C := A^T * B$.

- $m$ Number of rows of the matrix $A$.

- $n$ Number of columns of the matrix $A$.

- $k$ Number of columns of the matrix $B$.

- $a$ Array containing non-zero elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to values array description in Sparse Matrix Storage Formats for more details.

- $ja$ Array containing the column indices plus one for each non-zero element of the matrix $A$. For each row the column indices must be arranged in the increasing order.
  - The length of this array is equal to the length of the array $a$. Refer to columns array description in Sparse Matrix Storage Formats for more details.

- $ia$ Array of length $m + 1$ when $trans = \text{'N'}$ or $\text{'n'}$, or $n + 1$ otherwise.
  - This array contains indices of elements in the array $a$, such that $ia[i] - ia[0]$ is the index in the array $a$ of the first non-zero element from the row $i$. The value of the last element $ia[m]$ or $ia[n]$ is equal to the number of non-zero elements of the matrix $A$ plus one. Refer to rowIndex array description in Sparse Matrix Storage Formats for more details.

- $b$ Array containing non-zero elements of the matrix $B$. Its length is equal to the number of non-zero elements in the matrix $B$. Refer to values array description in Sparse Matrix Storage Formats for more details.

- $jb$ Array containing the column indices plus one for each non-zero element of the matrix $B$. For each row the column indices must be arranged in the increasing order.
The length of this array is equal to the length of the array \( b \). Refer to columns array description in Sparse Matrix Storage Formats for more details.

\( \text{ib} \)

Array of length \( m + 1 \).

This array contains indices of elements in the array \( b \), such that \( \text{ib}[i] - \text{ib}[0] \) is the index in the array \( b \) of the first non-zero element from the row \( i \). The value of the last element \( \text{ib}[m] \) is equal to the number of non-zero elements of the matrix \( B \) plus one. Refer to \( \text{rowIndex} \) array description in Sparse Matrix Storage Formats for more details.

Output Parameters

\( \text{c} \)

Array containing non-zero elements of the result matrix \( C \).

\( \text{ldc} \)

Specifies the leading dimension of the dense matrix \( C \) as declared in the calling (sub)program. Must be at least \( \max(m, 1) \) when \( \text{trans} = 'N' \) or \( 'n' \), or \( \max(1, n) \) otherwise.

**Sparse QR Routines**

Sparse QR routines and their data types

<table>
<thead>
<tr>
<th>Routine or function group</th>
<th>Data types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{mkl_sparse_set_qr_hint}</td>
<td></td>
<td>Enables a pivot strategy for an ill-conditioned matrix.</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr}</td>
<td>s,d</td>
<td>Calculates the solution of a sparse system of linear equations using QR factorization.</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr_reorder}</td>
<td></td>
<td>Performs reordering and symbolic analysis of the matrix ( A ).</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr_factorize}</td>
<td>s,d</td>
<td>Performs numerical factorization of the matrix ( A ).</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr_solve}</td>
<td>s,d</td>
<td>Solves the system ( A \cdot x = b ) using QR factorization of the matrix ( A ).</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr_qmult}</td>
<td>s,d</td>
<td>Performs ( x := Q^{-1} \cdot b ).</td>
</tr>
<tr>
<td>\text{mkl_sparse_?_qr_rsolve}</td>
<td>s,d</td>
<td>Performs ( x := R^{-1} \cdot b ).</td>
</tr>
</tbody>
</table>

**mkl_sparse_set_qr_hint**

Define the pivot strategy for further calls of \text{mkl_sparse_?_qr}.

**Syntax**

\[
\text{sparse_status_t mkl_sparse_set_qr_hint} (\text{sparse_matrix_t } A, \text{sparse_qr_hint_t } hint);
\]

**Include Files**

- \text{mkl_sparse_qr.h}

**Description**

You can use this routine to enable a pivot strategy in the case of an ill-conditioned matrix.

**Input Parameters**

\( A \)

Handle containing a sparse matrix in an internal data structure.
hint

Value specifying whether to use pivoting.

**NOTE** The only value currently supported is SPARSE_QR_WITH_PIVOTS, which enables the use of a pivot strategy for an ill-conditioned matrix.

**Return Values**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_STATUS_SUCCESS</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SPARSE_STATUS_NOT_INITIALIZED</td>
<td>The routine encountered an empty handle or matrix array.</td>
</tr>
<tr>
<td>SPARSE_STATUS_ALLOC_FAILED</td>
<td>Internal memory allocation failed.</td>
</tr>
<tr>
<td>SPARSE_STATUS_INVALID_VALUE</td>
<td>The input parameters contain an invalid value.</td>
</tr>
<tr>
<td>SPARSE_STATUS_EXECUTION_FAILED</td>
<td>Execution failed.</td>
</tr>
<tr>
<td>SPARSE_STATUS_INTERNAL_ERROR</td>
<td>An error in algorithm implementation occurred.</td>
</tr>
<tr>
<td>SPARSE_STATUS_NOT_SUPPORTED</td>
<td>The requested operation is not supported.</td>
</tr>
</tbody>
</table>

**mkl_sparse_?_qr**

*Calculates the solution of sparse systems of linear equations using QR decomposition.*

**Syntax**

```c
sparse_status_t mkl_sparse_d_qr ( sparse_operation_t operation, sparse_matrix_t A, struct matrix_descr descr, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b, MKL_INT ldb );

sparse_status_t mkl_sparse_s_qr ( sparse_operation_t operation, sparse_matrix_t A, struct matrix_descr descr, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b, MKL_INT ldb );
```

**Include Files**

- mkl_sparse_qr.h

**Description**

The mkl_sparse_?_qr routine calculates the solution of sparse systems of linear equations \( A*x = b \) using QR decomposition (\( A = QR \)).

**NOTE** Currently, mkl_sparse_?_qr supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.
Input Parameters

operation

Specifies the operation to perform.

ОUTЕ NOTE Currently, the only supported value is SPARSE_OPERATION_NON_TRANSPOSE (non-transpose case; that is, \( A^T x = b \) is solved).

\( A \)

Handle containing a sparse matrix in an internal data structure.

\( descr \)

Structure specifying sparse-matrix properties. Only the parameters listed here are currently supported.

\( type \)

Specifies the type of sparse matrix.

ОUTЕ NOTE Currently, the only supported value is SPARSE_MATRIX_TYPE_GENERAL (the matrix is processed as-is).

\( layout \)

Describes the storage scheme for the dense matrix:

<table>
<thead>
<tr>
<th>Layout Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_COLUMN_MAJOR</td>
<td>Storage of elements uses column-major layout.</td>
</tr>
<tr>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
<td>Storage of elements uses row-major layout.</td>
</tr>
</tbody>
</table>

\( x \)

Array with a size of at least \( rows \times cols \):

<table>
<thead>
<tr>
<th>( rows ) (number of rows in ( x ))</th>
<th>( cols ) (number of columns in ( x ))</th>
<th>( layout = SPARSE_LAYOUT_COLUMN_MAJOR )</th>
<th>( layout = SPARSE_LAYOUT_ROW_MAJOR )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( idx )</td>
<td>( columns )</td>
<td>( Number of columns in A )</td>
<td>( idx )</td>
</tr>
</tbody>
</table>

\( b \)

Array with a size of at least \( rows \times cols \):

<table>
<thead>
<tr>
<th>( rows ) (number of rows in ( b ))</th>
<th>( cols ) (number of columns in ( b ))</th>
<th>( layout = SPARSE_LAYOUT_COLUMN_MAJOR )</th>
<th>( layout = SPARSE_LAYOUT_ROW_MAJOR )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( ldb )</td>
<td>( columns )</td>
<td>( Number of columns in A )</td>
<td>( ldb )</td>
</tr>
</tbody>
</table>
Specifies the leading dimension of matrix $b$.

**Output Parameters**

$x$ Overwritten by the updated matrix $y$.

**Return Values**

- **SPARSE_STATUS_SUCCESS** The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED** The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED** Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE** The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED** Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR** An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED** The requested operation is not supported.

**mkl_sparse_qr_reorder**

Reordering step of SPARSE QR solver.

**Syntax**

```
sparse_status_t mkl_sparse_qr_reorder (sparse_matrix_t A, struct matrix_descr descr);
```

**Include Files**

- mkl_sparse_qr.h

**Description**

The `mkl_sparse_qr_reorder` routine performs ordering and symbolic analysis of matrix $A$.

**NOTE** Currently, `mkl_sparse_qr_reorder` supports only general structure and CSR format for the input matrix.

**Input Parameters**

- **$A$** Handle containing a sparse matrix in an internal data structure.
- **$descr$** Structure specifying sparse-matrix properties. Only the parameters listed here are currently supported.

**Return Values**

- **SPARSE_STATUS_SUCCESS** The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

**mkl_sparse_?_qr_factorize**

*Factorization step of the SPARSE QR solver.*

**Syntax**

sparse_status_t mkl_sparse_d_qr_factorize (sparse_matrix_t A, double *alt_values);
sparse_status_t mkl_sparse_s_qr_factorize (sparse_matrix_t A, float *alt_values);

**Include Files**

- mkl_sparse_qr.h

**Description**

The `mkl_sparse_?_qr_factorize` routine perform numerical factorization of matrix $A$.

*NOTE* Currently, `mkl_sparse_?_qr_factorize` supports only CSR format for the input matrix.

**Input Parameters**

- $A$ Handle containing a sparse matrix in an internal data structure.
- $alt\_values$ Array with alternative values. Must be the size of the non-zeros in the initial input matrix. When passed to the routine, these values will be used during the factorization step instead of the values stored in handle $A$.

**Return Values**

- SPARSE_STATUS_SUCCESS The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.
- SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
mkl_sparse_?_qr_solve
Solving step of the SPARSE QR solver.

Syntax

```c
sparse_status_t mkl_sparse_d_qr_solve ( sparse_operation_t operation, sparse_matrix_t A, double *alt_values, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b, MKL_INT ldb );

sparse_status_t mkl_sparse_s_qr_solve ( sparse_operation_t operation, sparse_matrix_t A, float *alt_values, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b, MKL_INT ldb );
```

Include Files

- mkl_sparse_qr.h

Description

The `mkl_sparse_?_qr_solve` routine computes the solution of sparse systems of linear equations $A \times x = b$.

**NOTE**
Currently, `mkl_sparse_?_qr_solve` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.
Alternative values are not supported and must be set to NULL.

Input Parameters

- `operation` Specifies the operation to perform.

  **NOTE** Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, $A \times x = b$ is solved).

- `A` Handle containing a sparse matrix in an internal data structure.

- `alt_values` Reserved for future use.

- `layout` Describes the storage scheme for the dense matrix:

<table>
<thead>
<tr>
<th>Storage of elements uses column-major layout.</th>
<th>Storage of elements uses row-major layout.</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPARSE_LAYOUT_COLUMN_MAJOR</code></td>
<td><code>SPARSE_LAYOUT_ROW_MAJOR</code></td>
</tr>
</tbody>
</table>

Execution failed.

An error in algorithm implementation occurred.

The requested operation is not supported.
Array with a size of at least \( rows \times cols \):

<table>
<thead>
<tr>
<th>layout</th>
<th>layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_COLUMN_MAJOR</td>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
</tr>
<tr>
<td><strong>rows</strong> (number of rows in ( x ))</td>
<td>( ldx )</td>
</tr>
<tr>
<td></td>
<td>Number of columns in ( A )</td>
</tr>
<tr>
<td><strong>cols</strong> (number of columns in ( x ))</td>
<td>( columns )</td>
</tr>
<tr>
<td></td>
<td>( ldx )</td>
</tr>
</tbody>
</table>

\( columns \)

\( ldx \)

Number of columns in matrix \( b \).

Specifies the leading dimension of matrix \( x \).

Array with a size of at least \( rows \times cols \):

<table>
<thead>
<tr>
<th>layout</th>
<th>layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_COLUMN_MAJOR</td>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
</tr>
<tr>
<td><strong>rows</strong> (number of rows in ( b ))</td>
<td>( ldb )</td>
</tr>
<tr>
<td></td>
<td>Number of columns in ( A )</td>
</tr>
<tr>
<td><strong>cols</strong> (number of columns in ( b ))</td>
<td>( columns )</td>
</tr>
<tr>
<td></td>
<td>( ldb )</td>
</tr>
</tbody>
</table>

\( ldb \)

Specifies the leading dimension of matrix \( b \).

**Output Parameters**

\( x \)

Contains the solution of system \( A \times x = b \).

**Return Values**

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.
mkl_sparse_?_qr_qmult
First stage of the solving step of the SPARSE QR solver.

Syntax

sparse_status_t mkl_sparse_d_qr_qmult ( sparse_operation_t operation, sparse_matrix_t A, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b, MKL_INT ldb );
sparse_status_t mkl_sparse_s_qr_qmult ( sparse_operation_t operation, sparse_matrix_t A, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b, MKL_INT ldb );

Include Files
• mkl_sparse_qr.h

Description
The mkl_sparse_?_qr_qmult routine computes multiplication of inversed matrix Q and right-hand side matrix b. This routine can be used to perform the solving step in two separate calls as an alternative to a single call of mkl_sparse_?_qr_solve.

NOTE Currently, mkl_sparse_?_qr_qmult supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

operation
Specifies the operation to perform.

NOTE Currently, the only supported value is
SPARSE_OPERATION_NON_TRANSPOSE (non-transpose case; that is, A*x = b is solved).

A
Handle containing a sparse matrix in an internal data structure.

layout
Describes the storage scheme for the dense matrix:

<table>
<thead>
<tr>
<th>SPARSE_LAYOUT_COLUMN_MAJOR</th>
<th>Storage of elements uses column-major layout.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
<td>Storage of elements uses row-major layout.</td>
</tr>
</tbody>
</table>

x
Array with a size of at least rows*cols:

<table>
<thead>
<tr>
<th>rows (number of rows in x)</th>
<th>layout = SPARSE_LAYOUT_COLUMN_MAJOR</th>
<th>layout = SPARSE_LAYOUT_ROW_MAJOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldx</td>
<td>Number of columns in A</td>
<td></td>
</tr>
</tbody>
</table>
**Number of columns in matrix \( b \).**

Specifies the leading dimension of matrix \( x \).

**Array with a size of at least \( \text{rows} \times \text{cols} \):**

<table>
<thead>
<tr>
<th>( \text{layout} = \text{SPARSE_LAYOUT_COLUMN_MAJOR} )</th>
<th>( \text{layout} = \text{SPARSE_LAYOUT_ROW_MAJOR} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{rows} ) (number of rows in ( b ))</td>
<td>( \text{ldb} )</td>
</tr>
<tr>
<td>Number of columns in ( A )</td>
<td></td>
</tr>
<tr>
<td>( \text{cols} ) (number of columns in ( b ))</td>
<td>( \text{columns} )</td>
</tr>
<tr>
<td>( \text{ldb} )</td>
<td>( \text{ldb} )</td>
</tr>
</tbody>
</table>

**Output Parameters**

\( x \)

Overwritten by the updated matrix \( x = Q^{-1}b \).

**Return Values**

- \( \text{SPARSE\_STATUS\_SUCCESS} \) The operation was successful.
- \( \text{SPARSE\_STATUS\_NOT\_INITIALIZED} \) The routine encountered an empty handle or matrix array.
- \( \text{SPARSE\_STATUS\_ALLOC\_FAILED} \) Internal memory allocation failed.
- \( \text{SPARSE\_STATUS\_INVALID\_VALUE} \) The input parameters contain an invalid value.
- \( \text{SPARSE\_STATUS\_EXECUTION\_FAILED} \) Execution failed.
- \( \text{SPARSE\_STATUS\_INTERNAL\_ERROR} \) An error in algorithm implementation occurred.
- \( \text{SPARSE\_STATUS\_NOT\_SUPPORTED} \) The requested operation is not supported.

**mkl_sparse_?_qr_rsolve**

Second stage of the solving step of the \texttt{SPARSE QR solver}. 

---

**cols** 

(number of columns in \( x \))

\( \text{columns} \)

\( \text{ldx} \)

**Bla and Sparse BLAS Routines**
Syntax

```c
sparse_status_t mkl_sparse_d_qr_rsolve ( sparse_operation_t operation, sparse_matrix_t A, sparse_layout_t layout, MKL_INT columns, double *x, MKL_INT ldx, const double *b, MKL_INT ldb );
sparse_status_t mkl_sparse_s_qr_rsolve ( sparse_operation_t operation, sparse_matrix_t A, sparse_layout_t layout, MKL_INT columns, float *x, MKL_INT ldx, const float *b, MKL_INT ldb );
```

Include Files

- `mkl_sparse_qr.h`

Description

The `mkl_sparse_?_qr_rsolve` routine computes the solution of \( A x = b \).

**NOTE** Currently, `mkl_sparse_?_qr_rsolve` supports only CSR format for the input matrix, non-transpose operation, and single right-hand side.

Input Parameters

- **operation**
  Specifies the operation to perform.

  **NOTE** Currently, the only supported value is `SPARSE_OPERATION_NON_TRANSPOSE` (non-transpose case; that is, \( A x = b \) is solved).

- **A**
  Handle containing a sparse matrix in an internal data structure.

- **layout**
  Describes the storage scheme for the dense matrix:
  - `SPARSE_LAYOUT_COLUMN_MAJOR` Storage of elements uses column-major layout.
  - `SPARSE_LAYOUT_ROW_MAJOR` Storage of elements uses row-major layout.

- **x**
  Array with a size of at least \( rows \cdot cols \):

<table>
<thead>
<tr>
<th>layout =</th>
<th>layout =</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPARSE_LAYOUT_COLUMN_MAJOR</code></td>
<td><code>SPARSE_LAYOUT_ROW_MAJOR</code></td>
</tr>
<tr>
<td>( rows ) (number of rows in ( x ))</td>
<td>( ldx )</td>
</tr>
<tr>
<td>( cols ) (number of columns in ( x ))</td>
<td>( columns )</td>
</tr>
</tbody>
</table>

- **columns**
  Number of columns in matrix \( b \).

- **ldx**
  Specifies the leading dimension of matrix \( x \).
Array with a size of at least $rows \times cols$:

<table>
<thead>
<tr>
<th>layout = SPARSE_LAYOUT_COLUMN_MAJOR</th>
<th>layout = SPARSE_LAYOUT_ROW_MAJOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>rows (number of rows in $b$)</td>
<td>$ldb$</td>
</tr>
<tr>
<td>cols (number of columns in $b$)</td>
<td>columns $\times ldb$</td>
</tr>
</tbody>
</table>

$ldb$

Specifies the leading dimension of matrix $b$.

**Output Parameters**

$x$

Contains the solution of the system $A \times x = b$.

**Return Values**

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.

**BLAS and LAPACK Compact Routines**

**Overview**

Many HPC applications rely on the application of BLAS and LAPACK operations on groups of very small matrices. While existing batch Intel® MKL BLAS routines already provide meaningful speedup over OpenMP* loops around BLAS operations for these sizes, another customization offers potential speedup by allocating matrices in a **SIMD-friendly format**, thus allowing for cross-matrix vectorization in the BLAS and LAPACK routines of the Intel® MKL called **Compact BLAS and LAPACK**.

The main idea behind these compact methods is to create true SIMD computations in which subgroups of matrices are operated on with kernels that abstractly appear as scalar kernels, while registers are filled by cross-matrix vectorization.

These are the BLAS/LAPACK compact functions:

- `mkl_?gemm_compact`
- `mkl_?trsm_compact`
• mkl_potrf_compact
• mkl_getrfnp_compact
• mkl_geqrf_compact
• mkl_getrinp_compact

The compact API provides additional service functions to refactor data. Because this capability is not specific to any particular BLAS or LAPACK operation, this data manipulation can be executed once for an application's data, allowing the entire program -- consisting of any number of BLAS and LAPACK operations for which compact kernels have been written -- to be performed on the compact data without any refactoring. For applications working on data in compact format, the packing function need not be used.

See "About the Compact Format" below for more details.

Along with this new data format, the API consists of two components:

• **BLAS and LAPACK Compact Kernels**: The first component of the API is a compact kernel that works on matrices stored in compact format.

• **Service Functions for the Compact Format**: The second component of the API is a compact service function allowing for data to be factored into and out of compact format. These are:
  
  • mkl_gepack_compact
  • mkl_igeunpack_compact
  • mkl_get_format_compact
  • mkl_get_size_compact

Note that there are some **Numerical Limitations** for the routines mentioned above.

**About the Compact Format**

In compact format, for calculations involving **real** precision, matrices are organized in packs of size \( V \), where \( V \) is the SIMD vector length of the underlying architecture. Each pack is a 3D-tensor with the matrix index incrementing the fastest. These packs are then loaded into registers and operated on using SIMD instructions.
The figure below demonstrates the packing of a set of four $3 \times 3$ real-precision matrices into compact format. The pack length for this example is $V = 2$, resulting in 2 compact packs.

**Interleaved Data for Compact BLAS and LAPACK**

<table>
<thead>
<tr>
<th>A111</th>
<th>A112</th>
<th>A113</th>
<th>A114</th>
</tr>
</thead>
<tbody>
<tr>
<td>A121</td>
<td>A122</td>
<td>A123</td>
<td>A124</td>
</tr>
<tr>
<td>A131</td>
<td>A132</td>
<td>A133</td>
<td>A134</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A211</th>
<th>A212</th>
<th>A213</th>
<th>A214</th>
</tr>
</thead>
<tbody>
<tr>
<td>A221</td>
<td>A222</td>
<td>A223</td>
<td>A224</td>
</tr>
<tr>
<td>A231</td>
<td>A232</td>
<td>A233</td>
<td>A234</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A311</th>
<th>A312</th>
<th>A313</th>
<th>A314</th>
</tr>
</thead>
<tbody>
<tr>
<td>A321</td>
<td>A322</td>
<td>A323</td>
<td>A324</td>
</tr>
<tr>
<td>A331</td>
<td>A332</td>
<td>A333</td>
<td>A334</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A411</th>
<th>A412</th>
<th>A413</th>
<th>A414</th>
</tr>
</thead>
<tbody>
<tr>
<td>A421</td>
<td>A422</td>
<td>A423</td>
<td>A424</td>
</tr>
<tr>
<td>A431</td>
<td>A432</td>
<td>A433</td>
<td>A434</td>
</tr>
</tbody>
</table>
For calculations involving complex precision, the real and imaginary parts of each matrix are packed separately. In the figure below, the group of four 3 x 3 complex matrices is packed into compact format with pack length $V = 2$. The first pack consists of the real parts of the first two matrices, and the second pack consists of the imaginary parts of the first two matrices. Real and imaginary packs alternate in memory. This storage format means that all compact arrays can be handled as a real type.

**Compact Format for Complex Precision**

The particular specifications (size and number) of the compact packs for the architecture and problem-precision definition are specified by an *MKL_COMPACT_PACK* enum type. For example: given a double-precision problem involving a group of 128 matrices working on an architecture with a 256-bit SIMD vector length, the optimal pack length is $V = 4$, and the number of packs is 32.

The initially-permitted values for the enum are:

- *MKL_COMPACT_SSE* - pack length 2 for double precision, pack length 4 for single precision.
- *MKL_COMPACT_AVX* - pack length 4 for double precision, pack length 8 for single precision.
- *MKL_COMPACT_AVX512* - pack length 8 for double precision, pack length 16 for single precision.

For calculations involving complex precision, the pack length is the same; however, half of the packs store the real parts of matrices, and half store the imaginary parts. The means that it takes double the number of packs to store the same number of matrices.
The above examples illustrate the case when the number of matrices is evenly-divisible by the pack length. When this is not the case, there will be partially-unfilled packs at the end of the memory segment, and the compact-packing routine will pad these partially unfilled packs with *identity matrices*, so that compact routines use only the completely-filled registers in their calculations. The next figure illustrates this padding for a group of three 3 x 3 real-precision matrices with a pack length of 2.

**Compact Format with Padding**

![Diagram showing compact format with padding](image)

Before calling a BLAS or LAPACK compact function, the input data must be packed in compact format. After execution, the output data should be unpacked from this compact format, unless another compact routine will be called immediately following the first. Two service functions, *mkl_?gepack_compact*, and *mkl_?geunpack_compact*, facilitate the process of storing matrices in compact format. It is recommended that the user call the function *mkl_get_format_compact* before calling the *mkl_?gepack_compact* routine to obtain the optimal format for performance. Advanced users can pack and unpack the matrices themselves and still use Intel® MKL compact functions on the packed set.

Compact routines can only be called for groups of matrices that have the same dimensions, leading dimension, and storage format. For example, the routine *mkl_?getrfnp_compact*, which calculates the LU factorization of a group of m x n matrices without pivoting, can only be called for a group of matrices with the same number of rows (m) and the same number of columns (n). All of the matrices must also be stored in arrays with the same leading dimension, and all must be stored in the same storage format (column-major or row-major).

### mkl_?gemm_compact

*Computes a matrix-matrix product of a set of compact format general matrices.*

**Syntax**

```c
void mkl_sgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb,
MKL_INT m, MKL_INT n, MKL_INT k, float alpha, const float *ap, MKL_INT ldap, const float
*bp, MKL_INT ldbp, float beta, float *cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT
nm);
```
void mkl_dgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb, MKL_INT m, MKL_INT n, MKL_INT k, double alpha, const double *ap, MKL_INT ldap, const double *bp, MKL_INT ldbp, double beta, double *cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_cgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb, MKL_INT m, MKL_INT n, MKL_INT k, mkl_compact_complex_float *alpha, const float *ap, MKL_INT ldap, const float *bp, MKL_INT ldbp, mkl_compact_complex_float *beta, float *cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_zgemm_compact (MKL_LAYOUT layout, MKL_TRANSPOSE transa, MKL_TRANSPOSE transb, MKL_INT m, MKL_INT n, MKL_INT k, mkl_compact_complex_double *alpha, const double *ap, MKL_INT ldap, const double *bp, MKL_INT ldbp, mkl_compact_complex_double *beta, double *cp, MKL_INT ldcp, MKL_COMPACT_PACK format, MKL_INT nm);

Description

The mkl_?gemm_compact routine computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for a group of nm general matrices $A_c$ that have been stored in compact format. The operation is defined for each matrix as:

$$C_c := \alpha \cdot \text{op}(A_c) \cdot \text{op}(B_c) + \beta \cdot C_c$$

Where

- $\text{op}(X_c)$ is one of $\text{op}(X_c) = X_c$, or $\text{op}(X_c) = X_c^T$, or $\text{op}(X_c) = X_c^H$,
- $\alpha$ and $\beta$ are scalars,
- $A_c$, $B_c$, and $C_c$ are matrices that have been stored in compact format,
- $\text{op}(A_c)$ is an $m$-by-$k$ matrix for each matrix in the group,
- $\text{op}(B_c)$ is a $k$-by-$n$ matrix for each matrix in the group,
- and $C_c$ is an $m$-by-$n$ matrix.

Input Parameters

layout

Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).

transa

Specifies the operation:

If $\text{transa}=\text{MKL_NOTRANS}$, then $\text{op}(A_c) := A_c$.
If $\text{transa}=\text{MKL_TRANS}$, then $\text{op}(A_c) := A_c^T$.
If $\text{transa}=\text{MKL_CONJTRANS}$, then $\text{op}(A_c) := A_c^H$.

transb

Specifies the operation:

If $\text{transb}=\text{MKL_NOTRANS}$, then $\text{op}(B_c) := B_c$.
If $\text{transb}=\text{MKL_TRANS}$, then $\text{op}(B_c) := B_c^T$.
If $\text{transb}=\text{MKL_CONJTRANS}$, then $\text{op}(B_c) := B_c^H$.

m

The number of rows of the matrices $\text{op}(A_c)$, $m \geq 0$.

n

The number of columns of matrices $\text{op}(B_c)$ and $C_c$, $n \geq 0$.

k

The number of columns of matrices $\text{op}(A_c)$ and the number of rows of matrices $\text{op}(B_c)$, $k \geq 0$.

alpha

Specifies the scalar $\alpha$. 
Points to the beginning of the array that stores the \( nmA_c \) matrices. See Compact Format for more details.

<table>
<thead>
<tr>
<th>( transa=\text{MKL_NOTRANS} ) &amp;</th>
<th>( transa=\text{MKL_TRANS or transa=MKL_CONJTRANS} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( layout = \text{MKL_COL_MAJOR} ) &amp; ( ap ) has size ( ldap\times k\times nm ).</td>
<td></td>
</tr>
<tr>
<td>( layout = \text{MKL_ROW_MAJOR} ) &amp; ( ap ) has size ( ldap\times m\times nm ).</td>
<td></td>
</tr>
</tbody>
</table>

Specifies the leading dimension of \( A_c \).

Points to the beginning of the array that stores the \( nmB_c \) matrices. See Compact Format for more details.

<table>
<thead>
<tr>
<th>( transb=\text{MKL_NOTRANS} ) &amp;</th>
<th>( transb=\text{MKL_TRANS or transb=MKL_CONJTRANS} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( layout = \text{MKL_COL_MAJOR} ) &amp; ( bp ) has size ( ldbp\times n\times nm ).</td>
<td></td>
</tr>
<tr>
<td>( layout = \text{MKL_ROW_MAJOR} ) &amp; ( bp ) has size ( ldbp\times k\times nm ).</td>
<td></td>
</tr>
</tbody>
</table>

Specifies the leading dimension of \( B_c \).

Specifies the scalar \( \beta \).

Before entry, \( cp \) points to the beginning of the array that stores the \( nmC_c \) matrices, except when \( \beta \) is equal to zero, in which case \( cp \) need not be set on entry.

|\( layout = \text{MKL\_COL\_MAJOR} \) & \( cp \) has size \( ldap\times n\times nm \). |
|\( layout = \text{MKL\_ROW\_MAJOR} \) & \( cp \) has size \( ldap\times m\times nm \). |

Specifies the leading dimension of \( C_c \).

|\( layout = \text{MKL\_COL\_MAJOR} \) & \( ldcp \) must be at least \( (1,m) \). |
|\( layout = \text{MKL\_ROW\_MAJOR} \) & \( ldcp \) must be at least \( (1,n) \). |

Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.

Total number of matrices stored in compact format in the group of matrices.
NOTE
The values of ldap, ldbp, and ldcp used in mkl_?gemm_compact must be consistent with the values used in mkl_?get_size_compact, mkl_?gepack_compact, and mkl_?geunpack_compact.

Output Parameters

cp
Each matrix \( C_c \) is overwritten by the \( m \)-by-\( n \) matrix \((alpha*op(A_c)*op(B_c) + beta*C_c)\).

**mkl_?trsm_compact**
Solves a triangular matrix equation for a set of general, \( m \times n \) matrices that have been stored in Compact format.

**Syntax**

```c
mkl_strsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, float alpha, const float *ap, MKL_INT a_stride, float *bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

```c
mkl_dtrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, double alpha, const double*ap, MKL_INT a_stride, double *bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

```c
mkl_ctrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, mkl_compact_complex_float *alpha, const float*ap, MKL_INT a_stride, float*bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

```c
mkl_ztrsm_compact (MKL_LAYOUT layout, MKL_SIDE side, MKL_UPLO uplo, MKL_TRANSPOSE transa, MKL_DIAG diag, MKL_INT m, MKL_INT n, mkl_compact_complex_double *alpha, const double*ap, MKL_INT a_stride, double*bp, MKL_INT b_stride, MKL_COMPACT_PACK format, MKL_INT nm);
```

**Description**
The routine solves one of the following matrix equations for a group of \( nm \) matrices:

\[
op(A_c) \ast X_c = \alpha \ast B_c\]

or

\[
X_c \ast \text{op}(A_c) = \alpha \ast B_c
\]

where:

- \( \alpha \) is a scalar, \( X_c \) and \( B_c \) are \( m \)-by-\( n \) matrices that have been stored in compact format, and \( A_c \) is a \( m \)-by-\( m \) unit, or non-unit, upper or lower triangular matrix that has been stored in compact format.
- \( \text{op}(A_c) \) is one of \( \text{op}(A_c) = A_c \), or \( \text{op}(A_c) = A_c^T \), or \( \text{op}(A_c) = A_c^H \),
- \( B_c \) is overwritten by the solution matrix \( X_c \).

**Input Parameters**

**layout**
Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
Specifies whether op(A_c) appears on the left or right of X_c in the equation:

if side = MKL_LEFT, then op(A_c)*X_c = alpha*B_c, if side = MKL_RIGHT, then X_c*op(A_c) = alpha*B_c

**uplo**

Specifies whether matrix A_c is upper or lower triangular.

If uplo = MKL_UPPER, A_c is upper triangular.

If uplo = MKL_LOWER, A_c is lower triangular.

**transa**

Specifies the operation:

If transa=MKL_NOTRANS, then op(A_c) = A_c;

If transa=MKL_TRANS, then op(A_c) = A_c^T;

If transa=MKL_CONJTRANS, then op(A_c) = A_c^H;

**diag**

Specifies whether the matrix A_c is unit triangular:

If diag=MKL_UNIT, then the matrix is unit triangular;

if diag=MKL_NONUNIT, then the matrix is not unit triangular.

**m**

The number of rows of B_c and the number of rows and columns of A_c when side=MKL_LEFT; m >= 0.

**n**

The number of columns of B_c and the number of rows and columns of A_c when side=MKL_RIGHT; n >= 0.

**alpha**

Specifies the scalar alpha. When alpha is zero, then ap is not referenced and bp need not be set before entry.

**ap**

Array, size ldap*k*nm, where k is m when side = MKL_LEFT and n when side = MKL_RIGHT. ap points to the beginning of nm A_c matrices stored in compact format. When uplo = MKL_UPPER, A_c is assumed to be an upper triangular matrix and the lower triangular part of A_c is not referenced. With uplo = MKL_LOWER, A_c is assumed to be a lower triangular matrix and the upper triangular part of A_c is not referenced. With diag = MKL_UNIT, the diagonal elements of A_c are not referenced either, but are assumed to be unity.

**ldap**

Column stride (column-major layout) or row stride (row-major layout) of A_c.

When side=MKL_LEFT, ldap must be at least max (1,m).

When side=MKL_RIGHT, ldap must be at least max (1,n).

**bp**

Array, size ldbp*n*nm when layout = MKL_COL_MAJOR; size ldbp*m*nm when layout = MKL_ROW_MAJOR. Before entry, bp points to the beginning of nm B_c matrices stored in compact format.

**ldbp**

Column stride (column-major layout) or row stride (row-major layout) of B_c.

<table>
<thead>
<tr>
<th>layout</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COL_MAJOR</td>
<td>ldbp must be at least max (1,m).</td>
</tr>
<tr>
<td>MKL_ROW_MAJOR</td>
<td>*ldbp must be at least max (1,n).</td>
</tr>
</tbody>
</table>

**format**

Specifies the format of the compact matrices. See <Compact Format> or mkl_get_format_compact for details.
Total number of matrices stored in Compact format; \( nm \geq 0 \).

**NOTE**
The values of `ldap` and `ldbp` used in `mkl_\text{?}trsm\text{?}compact` must be consistent with the values used in `mkl_\text{?}get\text{?}size\text{?}compact`, `mkl_\text{?}gepack\text{?}compact`, and `mkl_\text{?}geunpack\text{?}compact`.

### Output Parameters

\( bp \)

On exit, \( B_c \) is overwritten by the solution matrix \( X_c \). \( bp \) points to the beginning of \( nm \) such \( X_c \) matrices.

### mkl_\text{?}potrf\text{?}compact

**Computes the Cholesky factorization of a set of symmetric (Hermitian), positive-definite matrices, stored in Compact format (see Compact Format for details).**

**Syntax**

```c
void mkl_spotrf\text{?}compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, float * ap, MKL_INT \( \text{ldap} \), MKL_INT * info, MKL_COMPACT\text{?}PACK format, MKL_INT nm);

void mkl_cpotrf\text{?}compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, float * ap, MKL_INT \( \text{ldap} \), MKL_INT * info, MKL_COMPACT\text{?}PACK format, MKL_INT nm);

void mkl_dpotrf\text{?}compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, double * ap, MKL_INT \( \text{ldap} \), MKL_INT * info, MKL_COMPACT\text{?}PACK format, MKL_INT nm);

void mkl_zpotrf\text{?}compact (MKL_LAYOUT layout, MKL_UPLO uplo, MKL_INT n, double * ap, MKL_INT \( \text{ldap} \), MKL_INT * info, MKL_COMPACT\text{?}PACK format, MKL_INT nm);
```

**Description**

The routine forms the Cholesky factorization of a set of symmetric, positive definite (or, for complex data, Hermitian, positive-definite), \( n \times n \) matrices \( A_c \), stored in Compact format, as:

- \( A_c = U_c^T U_c \) (for real data), \( A_c = U_c^H U_c \) (for complex data), if `uplo = MKL\_UPPER`
- \( A_c = L_c^T L_c \) (for real data), \( A_c = L_c^H L_c \) (for complex data), if `uplo = MKL\_LOWER`

where \( L_c \) is a lower triangular matrix, and \( U_c \) is upper triangular. The factorization (output) data will also be stored in Compact format.

Before calling this routine, call `mkl_\text{?}gepack\text{?}compact` to store the matrices in the Compact format.

**NOTE**

Compact routines have some limitations; see Numerical Limitations.

### Input Parameters

\( layout \)

Specifies whether two-dimensional array storage is row-major (`MKL\_ROW\_MAJOR`) or column-major (`MKL\_COL\_MAJOR`).

\( uplo \)

Must be `MKL\_UPPER` or `MKL\_LOWER`.
Indicates whether the upper or lower triangular part of $A_c$ has been stored and will be factored.

If $\text{uplo} = \text{MKL\_UPPER}$, the upper triangular part of $A_c$ is stored, and the strictly lower triangular part of $A_c$ is not referenced.

If $\text{uplo} = \text{MKL\_LOWER}$, the lower triangular part of $A_c$ is stored, and the strictly upper triangular part of $A_c$ is not referenced.

$n$ The order of $A_c$; $n \geq 0$.

$ldap$ Column stride (column-major layout) or row stride (row-major layout) of $A_c$.

$ap$ Points to the beginning of the $nm$ $A_c$ matrices. On entry, $ap$ contains either the upper or the lower triangular part of $A_c$ (see uplo).

$format$ Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.

$nm$ Total number of matrices stored in Compact format; $nm \geq 0$.

Application Notes:

Before calling this routine, mkl_?gepack_compact must be called. After calling this routine, mkl_?geunpack_compact should be called, unless another compact routine will be called for the Compact format matrices.

The total number of floating-point operations is approximately $nm \times \frac{1}{3} n^3$ for real flavors and $nm \times \frac{4}{3} n^3$ for complex flavors.

Output Parameters

$ap$ The upper or lower triangular part of $A_c$, stored in Compact format in $ap$, is overwritten by its Cholesky factor $U_c$ or $L_c$ (as specified by $uplo$). $ap$ now points to the beginning of this set of factors, stored in Compact format.

$info$ The parameter is not currently used in this routine. It is reserved for the future use.

mkl_?getrfnp_compact

The routine computes the LU factorization, without pivoting, of a set of general, $m \times n$ matrices that have been stored in Compact format (see Compact Format).

Syntax

```c
void mkl_sgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_dgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_cgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_zgetrfnp_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT ldap, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```
Description

The `mkl_?getrfnp_compact` routine calculates the LU factorizations of a set of \( nm \) general \((m \times n)\) matrices \( A \), stored in Compact format, as \( A_c = L_c \cdot U_c \). The factorization (output) data will also be stored in Compact format.

**NOTE**
Compact routines have some limitations; see Numerical Limitations.

## Input Parameters

- **layout**
  Specifies whether two-dimensional array storage is row-major (`MKL_ROW_MAJOR`) or column-major (`MKL_COL_MAJOR`).
- **m**
  The number of rows of \( A \); \( m \geq 0 \).
- **n**
  The number of columns of \( A \); \( n \geq 0 \).
- **ap**
  Points to the beginning of the array which stores \( nm \) \( A_c \) matrices.
  See Compact Format for more details.
- **ldap**
  Leading dimension of \( A_c \).
- **format**
  Specifies the format of the compact matrices. See Compact Format or `mkl_get_format_compact` for details.
- **nm**
  Total number of matrices stored in Compact format.

**Application Notes:**

Before calling this routine, `mkl_?gepack_compact` must be called. After calling this routine, `mkl_?geunpack_compact` should be called, unless another compact routine will be subsequently called for the Compact format matrices.

The approximate number of floating-point operations for real flavors is:

- \( nm \cdot (2/3)n^3 \), if \( m = n \),
- \( nm \cdot (1/3)n^2(3m-n) \), if \( m > n \),
- \( nm \cdot (1/3)m^2(3n-m) \), if \( m < n \).

The number of operations for complex flavors is four times greater. Directly after calling this routine, you can call the following:

- `mkl_?getrinp_compact`, for computing the inverse of the \( nm \) input matrices in Compact format

## Output Parameters

- **ap**
  On exit, \( A_c \) is overwritten by its factorization data. \( ap \) points to the beginning of \( nm \) \( L_c \) and \( U_c \) factors of \( A_c \). The unit diagonal elements of \( L_c \) are not stored.
- **info**
  The parameter is not currently used in this routine. It is reserved for the future use.
**mkl_?geqrf_compact**

Computes the QR factorization of a set of general m x n, matrices, stored in Compact format (see Compact Format for details).

**Syntax**

```c
void mkl_sgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT ldap, float * taup, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_cgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, float * ap, MKL_INT ldap, float * taup, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_dgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT ldap, double * taup, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);

void mkl_zgeqrf_compact (MKL_LAYOUT layout, MKL_INT m, MKL_INT n, double * ap, MKL_INT ldap, double * taup, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```

**Description**

The routine forms the QR factorization of a set of general, m x n matrices A, stored in Compact format. The routine does not form the Q factors explicitly. Instead, Q is represented as a product of min(m,n) elementary reflectors. The factorization (output) data will also be stored in Compact format.

---

**NOTE**

Compact routines have some limitations; see Numerical Limitations.

**Input Parameters**

- **layout**
  
  Specifies whether two-dimensional array storage is row-major (MKL_Row_MAJOR) or column-major (MKL_Col_MAJOR).

- **m**
  
  The number of rows of A; m ≥ 0.

- **n**
  
  The number of columns of A; n ≥ 0.

- **ap**
  
  Points to the beginning of the nm A matrices. On entry, ap contains either the upper or the lower triangular part of A (see uplo).

- **ldap**
  
  Column stride (column-major layout) or row stride (row-major layout) of A.

- **work**
  
  Points to the beginning of the workspace array.

- **lwork**
  
  The size of the work array. If lwork = -1, a workspace query is assumed; the routine only calculates the optimal size of the work array and returns this value as the first entry of the work array.

- **format**
  
  Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.

- **nm**
  
  Total number of matrices stored in Compact format.

**Application Notes:**
The compact array that will store the elementary reflectors needs to be allocated before the routine is called and unpacked after. First, the routine mkl_?get_size_compact should be called, to determine the size of taup, and memory for taup should be allocated. After calling mkl_?geqrf_compact, taup stores the elementary reflectors in compact form, so should be unpacked using mkl_?geunpack_compact. See Compact Format for more details, or reference the example below. (Note: the following example is meant to demonstrate the calling sequence to allocate memory and unpack taup. All other parameters are assumed to be already set up before the sequence below is executed.)

```c
MKL_R_TYPE *tau_array[nm];
// ...
tau_buffer_size = mkl_?get_size_compact(min(m, n), 1, format, nm);
MKL_R_TYPE *tau_compact = (MKL_R_TYPE *)mkl_malloc(tau_buffer_size, 128);
mkl_?geqrf_compact(layout, m, n, a_compact, ldap, tau_compact, work, lwork, &info, format, nm);
// Note that here MKL_COL_MAJOR is used because tau is a 1-d array
mkl_?geunpack_compact(MKL_COL_MAJOR, min(m, n), 1, tau_array, min(m, n), tau_compact, min(m, n),
format, nm);
```

### Output Parameters

- **ap**
  
  On exit, A_c is overwritten by its factorization data. ap points to the beginning of nm factorizations of A_c, stored in Compact format. The factorization data is stored as follows: The elements on and above the diagonal contain the min( m, n )-by- n upper trapezoidal matrix R_c ( R_c is upper triangular if m ≥ n ); the elements below the diagonal, with tau_c, present the orthogonal matrix Q_c as a product of min( m, n ) elementary reflectors (see Orthogonal Factorizations: LAPACK Computational Routines). See Compact Format for more details.

- **taup**
  
  Points to the beginning of a set of the tau_c arrays, each of which has size min(m,n), stored in Compact format. tau_c contains scalars that define elementary reflectors for Q_c in its decomposition in a product of elementary reflectors. taup needs to be allocated by the user before calling this routine. See the application notes (below the description) for more details.

- **work[0]**
  
  On exit contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.

- **info**
  
  The parameter is not currently used in this routine. It is reserved for the future use.

### mkl_?getrinp_compact

**Computes the inverse of a set of LU-factorized general matrices, without pivoting, stored in the compact format** (see Compact Format for details).

### Syntax

```c
void mkl_sgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, float * ap, MKL_INT lda, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_dgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, double * ap, MKL_INT lda, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_cgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, float * ap, MKL_INT lda, float * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
void mkl_zgetrinp_compact (MKL_LAYOUT layout, MKL_INT n, double * ap, MKL_INT lda, double * work, MKL_INT lwork, MKL_INT * info, MKL_COMPACT_PACK format, MKL_INT nm);
```
Description

This routine computes the inverse \( \text{inv}(A_c) \) of a set of general, \( n \times n \) matrices \( A_c \), that have been stored in Compact format. The factorization (output) data will also be stored in Compact format.

**NOTE**
Compact routines have some limitations; see Numerical Limitations.

**Input Parameters**

- **layout**: Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).
- **n**: The order of \( A_c \); \( n \geq 0 \).
- **ap**: Points to the beginning of the \( nm \) \( A_c \) matrices. On entry, \( ap \) contains the LU factorizations of \( A_c \), stored in Compact format, as returned by mkl_\text{getrfnp}_\text{compact} : \( A_c = L_c U_c \).
  See Compact Format for more details.
- **ldap**: Column stride (column-major layout) or row stride (row-major layout) of \( A_c \).
- **work**: Points to the beginning of the work array.
- **lwork**: The size of the work array. If lwork = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array.
- **format**: Specifies the format of the compact matrices. See Compact Format or mkl_\text{get_format}_\text{compact} for details.
- **nm**: Total number of matrices stored in Compact format.

**Application Notes:**
Before calling this routine, mkl_\text{?gepack}_\text{compact} must be called. After calling this routine, mkl_\text{?geunpack}_\text{compact} should be called, unless another compact routine will be subsequently called on the Compact format matrices.

The total number of floating-point operations is approximately \( nm \times (4/3) n^3 \) for real flavors and \( nm \times (16/3) n^3 \) for complex flavors.

**Output Parameters**

- **ap**: On exit, \( A_c \) is overwritten by \( \text{inv}(A_c) \). \( ap \) points to the beginning of \( nm \) \( \text{inv}(A_c) \) matrices stored in Compact format.
- **work[0]**: On exit, work[0] contains the minimum value of lwork required for optimum performance. Use this lwork for subsequent runs.
- **info**: The parameter is not currently used in this routine. It is reserved for the future use.

**Numerical Limitations for Compact BLAS and Compact LAPACK Routines**
Compact routines are subject to a set of numerical limitations. They also skip most of the checks presented in regular BLAS and LAPACK routines in order to provide effective vectorization. The following limitations apply to at least one compact routine.
Complex division: BLAS and LAPACK compact routines rely on a naïve method for complex division that does not protect the solution against overflow, underflow, or loss of precision.

Error checking : the LAPACK compact routines skip error checking for performance reasons; therefore, the user is responsible for passing correct parameters. There are no checks for incorrect matrices (such as singular for LU, non-positive-definite for Cholesky) - it is always assumed that the algorithm for the input matrix can be completed without error.

No pivoting: the generic LU factorization routine, ?getrf, calculates the factorization using partial pivoting. However, because pivoting includes comparisons which cannot be effectively vectorized, only non-pivoting versions of LU mkl_?getrfnp_compact and Inverse from LU (mkl_?getrinp_compact) are provided as compact routines.

Matrices scaled near underflow/overflow: the LAPACK compact routines do not provide safe handling for values near underflow/overflow. This means that Compact routines may return incorrect results for such matrices. This limitation is related to compact routine for QR: mkl_?geqrf_compact.

It is the responsibility of the user to ensure that the input matrices can be factorized, inverted, and/or solved given these numerical limitations.

mkl_?get_size_compact

Returns the buffer size, in bytes, needed to pack data in Compact format.

Syntax

MKL_INT mkl_sget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
MKL_INT mkl_dget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
MKL_INT mkl_cget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);
MKL_INT mkl_zget_size_compact (MKL_INT ld, MKL_INT sd, MKL_COMPACT_PACK format, MKL_INT nm);

Description

The routine returns the buffer size, in bytes, required for mkl_?gepack_compact.

Input Parameters

ld
Leading dimension of the matrices in Compact format.

sd
Second dimension of the matrices in Compact format.

format
 Describes the compact packing format according to the MKL_COMPACT_PACK enum type.

nm
Total number of matrices to be packed in Compact format.

Application Notes:

Before calling this routine, mkl_?get_format_compact can be called to determine the optimal format.

After calling this routine and allocating the amount of memory indicated by size, the user can call mkl_?gepack_compact to pack the nm input matrices in Compact format.

Return Values

This function returns a value size.
The buffer size, in bytes, required by the packing function mkl_?gepack_compact.

**mkl_get_format_compact**

*Returns the optimal compact packing format for the architecture, needed for all compact routines.*

**Syntax**

MKL_COMPACT_PACK mkl_get_format_compact ();

**Description**

The routine returns the optimal compact packing format, which is an MKL_COMPACT_PACK type, for the current architecture. The optimal value of format is determined by the architecture’s vector-register length. format is a required parameter for any packing, unpacking, or BLAS/LAPACK compact routine. See Compact Format for details.

**Return Values**

The function returns a value format.

<table>
<thead>
<tr>
<th>format</th>
<th>format can be returned as any of the following three values. MKL_COMPACT_AVX512 is the optimal format value for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COMPACT_AVX512</td>
<td>• Intel® Advanced Vector Extensions 512 (Intel® AVX-512)-enabled processors.</td>
</tr>
<tr>
<td>MKL_COMPACT_AVX</td>
<td>• Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture)-enabled processors.</td>
</tr>
<tr>
<td>MKL_COMPACT_SSE</td>
<td>• Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups processors.</td>
</tr>
</tbody>
</table>

**Application Notes:**

After calling this routine, mkl_?get_size_compact can be called to calculate the buffer size needed for mkl_?gepack_compact.

**mkl_?gepack_compact**

*Packs matrices from standard (row or column-major) format to Compact format.*
Syntax

mkl_sgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const float * const *a, MKL_INT lda, float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
mkl_dgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const double * const *a, MKL_INT lda, double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
mkl_cgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const mkl_compact_complex_float * const *a, MKL_INT lda, float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, const MKL_INT nm);
mkl_zgepack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, const mkl_compact_complex_double * const *a, MKL_INT lda, double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);

Description

The routine packs nm matrices A from standard format (row or column-major, pointer to pointer) in a into Compact format, storing the new compact format matrices Ac in array ap.

Input Parameters

- **layout**
  Specifies whether two-dimensional array storage is row-major (MKL_ROW_MAJOR) or column-major (MKL_COL_MAJOR).

- **rows**
  The number of rows of A; rows >= 0.

- **columns**
  The number of columns of A; columns >= 0.

- **a**
  A standard format (row or column-major, pointer-to-pointer) array, storing nm input A matrices.

- **lda**
  Leading dimension of A.

<table>
<thead>
<tr>
<th>layout</th>
<th>lda must be at least max (1,rows).</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COL_MAJOR</td>
<td></td>
</tr>
<tr>
<td>MKL_ROW_MAJOR</td>
<td></td>
</tr>
</tbody>
</table>

- **ldap**
  Leading dimension of Ac.

<table>
<thead>
<tr>
<th>layout</th>
<th>ldap must be at least max (1,columns).</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COL_MAJOR</td>
<td></td>
</tr>
<tr>
<td>MKL_ROW_MAJOR</td>
<td></td>
</tr>
</tbody>
</table>

- **format**
  Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.

- **nm**
  Total number of matrices that will be stored in Compact format.

Application Notes:

**NOTE**
The values of ldap used in mkl ?gepack_compact must be consistent with the values used in mkl ?get_size_compact and mkl ?geunpack_compact.
Directly after calling this routine, any BLAS or LAPACK compact routine can be called. Unpacking matrices from Compact format can be done by calling `mkl_?geunpack_compact`.

**Output Parameters**

`ap`  
Array storing the compact format input matrices $A_c$. $ap$ must have size at least $size = mkl_?get_size_compact$.

---

**mkl_?geunpack_compact**

*Unpacks matrices from Compact format to standard (row- or column-major, pointer-to-pointer) format.*

**Syntax**

```c
mkl_sgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, float * const *a, MKL_INT lda, const float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
mkl_dgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, double * const *a, MKL_INT lda, const double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
mkl_cgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, mkl_compact_complex_float * const *a, MKL_INT lda, const float *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
mkl_zgeunpack_compact (MKL_LAYOUT layout, MKL_INT rows, MKL_INT columns, mkl_compact_complex_double * const *a, MKL_INT lda, const double *ap, MKL_INT ldap, MKL_COMPACT_PACK format, MKL_INT nm);
```

**Description**

The routine unpacks $nm$ Compact format matrices $A_c$ from array $ap$ into standard (row- or column-major, pointer-to-pointer) format in array $A$.

**Input Parameters**

- `layout`  
  Specifies whether two-dimensional array storage is *row-major* (MKL_ROW_MAJOR) or *column-major* (MKL_COL_MAJOR).

- `rows`  
  The number of rows of $A$; $rows \geq 0$.

- `columns`  
  The number of columns of $A$; $columns \geq 0$.

- `lda`  
  Leading dimension of $A$.

<table>
<thead>
<tr>
<th><code>layout</code></th>
<th>±</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COL_MAJOR</td>
<td>lda must be at least max (1,rows).</td>
</tr>
<tr>
<td>MKL_ROW_MAJOR</td>
<td>lda must be at least max (1,columns).</td>
</tr>
</tbody>
</table>

- `ap`  
  Array storing the compact format of input matrices $A_c$. See [Compact Formator](#) `mkl_get_format_compact` for details.

<table>
<thead>
<tr>
<th><code>layout</code></th>
<th>±</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_COL_MAJOR</td>
<td>$ap$ has size $ldap<em>columns</em>nm$.</td>
</tr>
<tr>
<td>MKL_ROW_MAJOR</td>
<td>$ap$ has size $ldap<em>rows</em>nm$.</td>
</tr>
</tbody>
</table>
ldap

Leading dimension of of \( A_c \).

<table>
<thead>
<tr>
<th>layout = MKL_COL_MAJOR</th>
<th>ldap must be at least max (1, rows).</th>
</tr>
</thead>
<tbody>
<tr>
<td>layout = MKL_ROW_MAJOR</td>
<td>ldap must be at least max (1, columns).</td>
</tr>
</tbody>
</table>

**NOTE**
The values of ldap used in mkl_?geunpack_compact must be consistent with the values used in mkl_?get_size_compact and mkl_?gepack_compact.

format

Specifies the format of the compact matrices. See Compact Format or mkl_get_format_compact for details.

nm

Total number of matrices that will be stored in Compact format.

**Output Parameters**

\( a \)

A standard format (row- or column-major, pointer-to-pointer) array, storing \( nm \) output \( A \) matrices.

<table>
<thead>
<tr>
<th>layout = MKL_COL_MAJOR</th>
<th>( a ) has size ( lda \times \text{columns} \times nm ).</th>
</tr>
</thead>
<tbody>
<tr>
<td>layout = MKL_ROW_MAJOR</td>
<td>( a ) has size ( lda \times \text{rows} \times nm ).</td>
</tr>
</tbody>
</table>

**Inspector-executor Sparse BLAS Routines**

The inspector-executor API for Sparse BLAS divides operations into two stages: analysis and execution. During the initial analysis stage, the API inspects the matrix sparsity pattern and applies matrix structure changes. In the execution stage, subsequent routine calls reuse this information in order to improve performance.

The inspector-executor API supports key Sparse BLAS operations for iterative sparse solvers:

- Sparse matrix-vector multiplication
- Sparse matrix-matrix multiplication with a sparse or dense result
- Solution of triangular systems
- Sparse matrix addition

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
### Naming conventions in Inspector-executor Sparse BLAS Routines

The Inspector-executor Sparse BLAS API routine names use the following convention:

\[
mkl\_sparse\_[<\text{character}>]_<\text{operation}>\_[<\text{format}>]
\]

The `<character>` field indicates the data type:

- `s`: real, single precision
- `c`: complex, single precision
- `d`: real, double precision
- `z`: complex, double precision

The data type is included in the name only if the function accepts dense matrix or scalar floating point parameters.

The `<operation>` field indicates the type of operation:

- `create`: create matrix handle
- `copy`: create a copy of matrix handle
- `convert`: convert matrix between sparse formats
- `export`: export matrix from internal representation to CSR or BSR format
- `destroy`: frees memory allocated for matrix handle
- `set_<op>_hint`: provide information about number of upcoming compute operations and operation type for optimization purposes, where `<op>` is `mv`, `sv`, `mm`, `sm`, `dotmv`, `symgs`, or `memory`
- `optimize`: analyze the matrix using hints and store optimization information in matrix handle
- `mv`: compute sparse matrix-vector product
- `mm`: compute sparse matrix by dense matrix product (batch mv)
- `set_value`: change a value in a matrix
- `spmm/spmd`: compute sparse matrix by sparse matrix product and store the result as a sparse/dense matrix
- `trsv`: solve a triangular system
- `trsm`: solve a triangular system with multiple right-hand sides
- `add`: compute sum of two sparse matrices
- `symgs`: compute a symmetric Gauss-Zeidel preconditioner
- `symgs_mv`: compute a symmetric Gauss-Zeidel preconditioner with a final matrix-vector multiplication
- `sypr`: compute the symmetric or Hermitian product of sparse matrices and store the result as a sparse matrix
- `syprd`: compute the symmetric or Hermitian product of sparse and dense matrices and store the result as a dense matrix
- `syrk`: compute the product of sparse matrix with its transposed matrix and store the result as a sparse matrix
syrkd: compute the product of sparse matrix with its transposed matrix and store the result as a dense matrix.

order: perform ordering of column indexes of the matrix in CSR format.

dotmv: compute a sparse matrix-vector product with dot product.

The `<format>` field indicates the sparse matrix storage format:

- coo: coordinate format
- bsr: block sparse row format plus variations. Fill out either `rows_start` and `rows_end` (for 4-arrays representation) or `rowIndex` array (for 3-array BSR/CSR).
- csr: compressed sparse row format plus variations. Fill out either `rows_start` and `rows_end` (for 4-arrays representation) or `rowIndex` array (for 3-array BSR/CSR).
- csc: compressed sparse column format plus variations. Fill out either `cols_start` and `cols_end` (for 4-arrays representation) or `colIndex` array (for 3-array CSC).

The format is included in the function name only if the function parameters include an explicit sparse matrix in one of the conventional sparse matrix formats.

**Sparse Matrix Storage Formats for Inspector-executor Sparse BLAS Routines**

Inspector-executor Sparse BLAS routines support four conventional sparse matrix storage formats:

- compressed sparse row format (CSR) plus variations
- compressed sparse column format (CSC) plus variations
- coordinate format (COO)
- block sparse row format (BSR) plus variations

Computational routines operate on a matrix handle that stores a matrix in CSR or BSR formats. Other formats should be converted to CSR or BSR format before calling any computational routines. For more information see [Sparse Matrix Storage Formats](#).

**Supported Inspector-executor Sparse BLAS Operations**

The Inspector-executor Sparse BLAS API can perform several operations involving sparse matrices. These notations are used in the description of the operations:

- $A$, $G$, $V$ are sparse matrices
- $B$ and $C$ are dense matrices
- $x$ and $y$ are dense vectors
- $alpha$ and $beta$ are scalars

$op(A)$ represents a possible transposition of matrix $A$

- $op(A) = A$
- $op(A) = A^T$ - transpose of $A$
- $op(A) = A^H$ - conjugate transpose of $A$

$op(A)^{-1}$ denotes the inverse of $op(A)$.

The Inspector-executor Sparse BLAS routines support the following operations:

- computing the vector product between a sparse matrix and a dense vector:
  
  $y := alpha*op(A)*x + beta*y$

- solving a single triangular system:
  
  $y := alpha*inv(op(A))*x$
• computing a product between a sparse matrix and a dense matrix:

\[ C := \alpha \cdot \text{op}(A) \cdot B + \beta \cdot C \]

• computing a product between sparse matrices with a sparse result:

\[ V := \alpha \cdot \text{op}(A) \cdot \text{op}(G) \]

• computing a product between sparse matrices with a dense result:

\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(G) \]

• computing a sum of sparse matrices with a sparse result:

\[ V := \alpha \cdot \text{op}(A) + G \]

• solving a sparse triangular system with multiple right-hand sides:

\[ C := \alpha \cdot \text{inv} \left( \text{op}(A) \right) \cdot B \]

Two-stage Algorithm for Inspector-executor Sparse BLAS routines

You can use a two-stage algorithm for Inspector-executor Sparse BLAS routines which produce a sparse matrix. The applicable routines are:

- `mkl_sparse_sp2m` (BSR/CSR/CSC formats)
- `mkl_sparse_sypr` (CSR format)
- `mkl_sparse_syrk` (CSR format) through the `mkl_sparse_sypr` interface.

In the two-stage algorithm:

1. The first stage constructs the structure of the output matrix.
   - For the BSR/CSR storage formats, fill out `rows_start` and either `rows_end` or `rowIndex` arrays for 4 or 3 array.
   - For the CSC storage format, fill out `cols_start` and either `cols_end` or `colIndex` arrays for 4 or 3 array.

   This stage also allows the user to estimate memory required for the desired operation.

2. The second stage constructs other arrays and performs the desired operation.

You can separate the calls for each stage. You can also perform the entire computation in a single call using the `sparse_request_t` parameter:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPARSE_STAGE_NNZ_COUNT</code></td>
<td>In the first stage, the algorithm computes only the row (CSR/BSR format) or column (CSC format) pointer array of the matrix storage format. The computed number of non-zeroes in the output matrix helps to calculate the amount of memory required.</td>
</tr>
<tr>
<td><code>SPARSE_STAGE_FINALIZE_MULT</code></td>
<td>In the second stage, the algorithm computes the remaining column (CSR/BSR format) or row (CSC format) index and value arrays for the output matrix. Use this value only after calling the function with <code>SPARSE_STAGE_NNZ_COUNT</code> first.</td>
</tr>
<tr>
<td><code>SPARSE_STAGE_FULL_MULT</code></td>
<td>Combine the two stages by performing the entire computation in a single step.</td>
</tr>
</tbody>
</table>

This example uses the two-stage algorithm for `mkl_sparse_sp2m` routine with a matrix in CSR format:

**First stage** (`sparse_request_t = SPARSE_STAGE_NNZ_COUNT`)

1. The algorithm calls the `mkl_sparse_sp2m` routine with the request parameter set to `SPARSE_STAGE_NNZ_COUNT`.
2. The algorithm exports the computed `rows_start` and `rows_end` arrays using the `mkl_sparse_x_export_csr` routine.
3. These arrays are used to calculate the number of non-zeroes (nnz) of the resulting output matrix.
Note that at this stage, the arrays related to column index and values for the output matrix have not been computed.

```c
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_NNZ_COUNT, &csrC);
/* optional calculation of nnz of resulting output matrix for computing memory requirement */
status = mkl_sparse_x_export_csr (csrC, &indexing, &rows, &cols, &rows_start, &rows_end, &col_indx, &values);
MKL_INT nnz = rows_end[rows-1] - rows_start[0];
```

**Second stage (sparse_request_t = SPARSE_STAGE_FINALIZE_MULT)**

The algorithm computes the remaining storage arrays (related to column index and values for the output matrix) and performs the desired operation.

```c
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FINALIZE_MULT, &csrC);
```

Alternatively, you can perform both operations in a single step:

**Single stage operation (sparse_request_t = SPARSE_STAGE_FULL_MULT)**

```c
status = mkl_sparse_sp2m (opA, descrA, csrA, opB, descrB, csrB, SPARSE_STAGE_FULL_MULT, &csrC);
```

---

**Matrix Manipulation Routines**

The Matrix Manipulation Routines table lists the matrix manipulation routines and the data types associated with them.

**Matrix Manipulation Routines and Their Data Types**

<table>
<thead>
<tr>
<th>Routine or Function Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_sparse_create_csr</td>
<td>s, d, c, z</td>
<td>Creates a handle for a CSR-format matrix.</td>
</tr>
<tr>
<td>mkl_sparse_create_csc</td>
<td>s, d, c, z</td>
<td>Creates a handle for a CSC format matrix.</td>
</tr>
<tr>
<td>mkl_sparse_create_coo</td>
<td>s, d, c, z</td>
<td>Creates a handle for a matrix in COO format.</td>
</tr>
<tr>
<td>mkl_sparse_create_bsr</td>
<td>s, d, c, z</td>
<td>Creates a handle for a matrix in BSR format.</td>
</tr>
<tr>
<td>mkl_sparse_copy</td>
<td>NA</td>
<td>Creates a copy of a matrix handle.</td>
</tr>
<tr>
<td>mkl_sparse_destroy</td>
<td>NA</td>
<td>Frees memory allocated for matrix handle.</td>
</tr>
<tr>
<td>mkl_sparse_convert_csr</td>
<td>NA</td>
<td>Converts internal matrix representation to CSR format.</td>
</tr>
<tr>
<td>mkl_sparse_convert_bsr</td>
<td>NA</td>
<td>Converts internal matrix representation to BSR format or changes BSR block size.</td>
</tr>
<tr>
<td>mkl_sparse_export_csr</td>
<td>s, d, c, z</td>
<td>Exports CSR matrix from internal representation.</td>
</tr>
<tr>
<td>mkl_sparse_export_csc</td>
<td>s, d, c, z</td>
<td>Exports CSC matrix from internal representation.</td>
</tr>
</tbody>
</table>
### Routine or Function Group | Data Types | Description
--- | --- | ---
mkl_sparse_?_export_bsr  | s, d, c, z  | Exports BSR matrix from internal representation.
mkl_sparse_?_set_value  | s, d, c, z  | Changes a single value of matrix in internal representation.
mkl_sparse_?_update_values  | s, d, c, z  | Changes all or selected matrix values in internal representation.
mkl_sparse_order  | NA  | Performs ordering of column indexes of the matrix in CSR format.

#### mkl_sparse_?_create_csr

Creates a handle for a CSR-format matrix.

**Syntax**

```c
sparse_status_t mkl_sparse_s_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, float *values);
sparse_status_t mkl_sparse_d_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, double *values);
sparse_status_t mkl_sparse_c_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex8 *values);
sparse_status_t mkl_sparse_z_create_csr (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex16 *values);
```

**Include Files**

- mkl_spblas.h

**Description**

The `mkl_sparse_?_create_csr` routine creates a handle for an \( m \)-by-\( k \) matrix \( A \) in CSR format.

**NOTE**

The input arrays provided are left unchanged except for the call to `mkl_sparse_order`, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use `mkl_sparse_copy`.

**Input Parameters**

- **indexing**
  
  Indicates how input arrays are indexed.

  - **SPARSE_INDEX_BASE_ZERO**  
    Zero-based (C-style) indexing: indices start at 0.

  - **SPARSE_INDEX_BASE_ONE**  
    One-based (Fortran-style) indexing: indices start at 1.
Number of rows of matrix A.

Number of columns of matrix A.

Array of length at least \( m \). This array contains row indices, such that \( \text{rows}_\text{start}[i] \) - indexing is the first index of row \( i \) in the arrays \( \text{values} \) and \( \text{col}_\text{indx} \). The value of indexing is 0 for zero-based indexing and 1 for one-based indexing.

Refer to \( \text{pointerB} \) array description in CSR Format for more details.

Array of at least length \( m \). This array contains row indices, such that \( \text{rows}_\text{end}[i] \) - indexing - 1 is the last index of row \( i \) in the arrays \( \text{values} \) and \( \text{col}_\text{indx} \). The value of indexing is 0 for zero-based indexing and 1 for one-based indexing.

Refer to \( \text{pointerE} \) array description in CSR Format for more details.

For one-based indexing, array containing the column indices plus one for each non-zero element of the matrix A. For zero-based indexing, array containing the column indices for each non-zero element of the matrix A. Its length is at least \( \text{rows}_\text{end}[\text{rows} - 1] \) - indexing.

The value of indexing is 0 for zero-based indexing and 1 for one-based indexing.

Array containing non-zero elements of the matrix A. Its length is equal to length of the \( \text{col}_\text{indx} \) array.

Refer to \( \text{values} \) array description in CSR Format for more details.

**Output Parameters**

**A** Handle containing internal data for subsequent Inspector-executor Sparse BLAS operations.

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

**SPARSE_STATUS_SUCCESS** The operation was successful.

**SPARSE_STATUS_NOT_INITIALIZED** The routine encountered an empty handle or matrix array.

**SPARSE_STATUS_ALLOC_FAILED** Internal memory allocation failed.

**SPARSE_STATUS_INVALID_VALUE** The input parameters contain an invalid value.

**SPARSE_STATUS_EXECUTION_FAILED** Execution failed.

**SPARSE_STATUS_INTERNAL_ERROR** An error in algorithm implementation occurred.

**SPARSE_STATUS_NOT_SUPPORTED** The requested operation is not supported.

**mkl_sparse_?_create_csc**

*Creates a handle for a CSC format matrix.*
Syntax

sparse_status_t mkl_sparse_s_create_csc (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT *cols_end, MKL_INT *row_indx, float *values);
sparse_status_t mkl_sparse_d_create_csc (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT *cols_end, MKL_INT *row_indx, double *values);
sparse_status_t mkl_sparse_c_create_csc (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT *cols_end, MKL_INT *row_indx, MKL_Complex8 *values);
sparse_status_t mkl_sparse_z_create_csc (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, MKL_INT *cols_start, MKL_INT *cols_end, MKL_INT *row_indx, MKL_Complex16 *values);

Include Files

• mkl_spblas.h

Description

The mkl_sparse_?_create_csc routine creates a handle for an $m$-by-$k$ matrix $A$ in CSC format.

NOTE

The input arrays provided are left unchanged except for the call to mkl_sparse_order, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use mkl_sparse_copy.

Input Parameters

indexing

Indicates how input arrays are indexed.

SPARSE_INDEX_BASE_ZERO

Zero-based (C-style) indexing: indices start at 0.

SPARSE_INDEX_BASE_ONE

One-based (Fortran-style) indexing: indices start at 1.

rows

Number of rows of the matrix $A$.

cols

Number of columns of the matrix $A$.

cols_start

Array of length at least $m$. This array contains col indices, such that cols_start[i] - ind is the first index of col $i$ in the arrays values and row_indx.ind takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to pointerB array description in CSC Format for more details.

cols_end

Array of at least length $m$. This array contains col indices, such that cols_end[i] - ind - 1 is the last index of col $i$ in the arrays values and row_indx.ind takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to pointerE array description in CSC Format for more details.
row_indx

For one-based indexing, array containing the row indices plus one for each non-zero element of the matrix A. For zero-based indexing, array containing the row indices for each non-zero element of the matrix A. Its length is at least \( cols \) \( \text{cols} - 1 \) \( \text{ind} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

values

Array containing non-zero elements of the matrix A. Its length is equal to length of the row_indx array.

Refer to values array description in CSC Format for more details.

Output Parameters

A

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_?_create_coo

Creates a handle for a matrix in COO format.

Syntax

sparse_status_t mkl_sparse_s_create_coo (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row indx, MKL_INT * col indx, float *values);

sparse_status_t mklsparse_d_create_coo (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row indx, MKL_INT * col indx, double *values);

sparse_status_t mkl_sparse_c_create_coo (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row indx, MKL_INT * col indx, MKL_Complex8 *values);

sparse_status_t mkl_sparse_z_create_coo (sparse_matrix_t *A, const sparse_index_base_t indexing, const MKL_INT rows, const MKL_INT cols, const MKL_INT nnz, MKL_INT *row indx, MKL_INT * col indx, MKL_Complex16 *values);

Include Files

• mkl_spblas.h
Description

The `mkl_sparse_?_create_coo` routine creates a handle for an \( m \)-by-\( k \) matrix \( A \) in COO format.

**NOTE**
The input arrays provided are left unchanged except for the call to `mkl_sparse_order`, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use `mkl_sparse_copy`.

**Input Parameters**

- **indexing**
  Indicating how input arrays are indexed.
  - `SPARSE_INDEX_BASE_ZERO` Zero-based (C-style) indexing: indices start at 0.
  - `SPARSE_INDEX_BASE_ONE` One-based (Fortran-style) indexing: indices start at 1.

- **rows**
  Number of rows of matrix \( A \).

- **cols**
  Number of columns of matrix \( A \).

- **nnz**
  Specifies the number of non-zero elements of the matrix \( A \).
  Refer to `nnz` description in Coordinate Format for more details.

- **row_indx**
  Array of length \( nnz \), containing the row indices for each non-zero element of matrix \( A \).
  Refer to `rows` array description in Coordinate Format for more details.

- **col_indx**
  Array of length \( nnz \), containing the column indices for each non-zero element of matrix \( A \).
  Refer to `columns` array description in Coordinate Format for more details.

- **values**
  Array of length \( nnz \), containing the non-zero elements of matrix \( A \) in arbitrary order.
  Refer to `values` array description in Coordinate Format for more details.

**Output Parameters**

- **A**
  Handle containing internal data.

**Return Values**
The function returns a value indicating whether the operation was successful or not, and why.

- `SPARSE_STATUS_SUCCESS` The operation was successful.
- `SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.
- `SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.
- `SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.
- `SPARSE_STATUS_EXECUTION_FAILED` Execution failed.
SPARSE_STATUS_INTERNAL_ERROR  An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

**mkl_sparse_?_create_bsr**

*Creates a handle for a matrix in BSR format.*

**Syntax**

```c
sparse_status_t mkl_sparse_s_create_bsr (sparse_matrix_t *A, const sparse_index_base_t indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols, const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, float *values);

sparse_status_t mkl_sparse_d_create_bsr (sparse_matrix_t *A, const sparse_index_base_t indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols, const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, double *values);

sparse_status_t mkl_sparse_c_create_bsr (sparse_matrix_t *A, const sparse_index_base_t indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols, const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex8 *values);

sparse_status_t mkl_sparse_z_create_bsr (sparse_matrix_t *A, const sparse_index_base_t indexing, const sparse_layout_t block_layout, const MKL_INT rows, const MKL_INT cols, const MKL_INT block_size, MKL_INT *rows_start, MKL_INT *rows_end, MKL_INT *col_indx, MKL_Complex16 *values);
```

**Include Files**

- mkl_spblas.h

**Description**

The `mkl_sparse_?_create_bsr` routine creates a handle for an \( m \)-by-\( k \) matrix \( A \) in BSR format.

**NOTE**

The input arrays provided are left unchanged except for the call to `mkl_sparse_order`, which performs ordering of column indexes of the matrix. To avoid any changes to the input data, use `mkl_sparse_copy`.

**Input Parameters**

- **indexing**
  Indicates how input arrays are indexed.
  - SPARSE_INDEX_BASE_ZERO  Zero-based (C-style) indexing: indices start at 0.
  - SPARSE_INDEX_BASE_ONE  One-based (Fortran-style) indexing: indices start at 1.

- **block_layout**
  Specifies layout of blocks:
  - SPARSE_LAYOUT_ROW_MAJ  Storage of elements of blocks uses row major layout.
Storage of elements of blocks uses column major layout.

**rows**

Number of block rows of matrix A.

**cols**

Number of block columns of matrix A.

**block_size**

Size of blocks in matrix A.

**rows_start**

Array of length \( m \). This array contains row indices, such that \( \text{rows_start}[i] - \text{ind} \) is the first index of block row \( i \) in the arrays \( \text{values} \) and \( \text{col_indx} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to pointerB array description in CSR Format for more details.

**rows_end**

Array of length \( m \). This array contains row indices, such that \( \text{rows_end}[i] - \text{ind} - 1 \) is the last index of block row \( i \) in the arrays \( \text{values} \) and \( \text{col_indx} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to pointerE array description in CSR Format for more details.

**col_indx**

For one-based indexing, array containing the column indices plus one for each non-zero block of the matrix A. For zero-based indexing, array containing the column indices for each non-zero block of the matrix A. Its length is \( \text{rows_end}[\text{rows} - 1] - \text{ind} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

**values**

Array containing non-zero elements of the matrix A. Its length is equal to length of the \( \text{col_indx} \) array multiplied by \( \text{block_size} \times \text{block_size} \).

Refer to the \( \text{values} \) array description in BSR Format for more details.

**Output Parameters**

**A**

Handle containing internal data.

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**
  The operation was successful.

- **SPARSE_STATUS_NOT_INITIALIZED**
  The routine encountered an empty handle or matrix array.

- **SPARSE_STATUS_ALLOC_FAILED**
  Internal memory allocation failed.

- **SPARSE_STATUS_INVALID_VALUE**
  The input parameters contain an invalid value.

- **SPARSE_STATUS_EXECUTION_FAILED**
  Execution failed.

- **SPARSE_STATUS_INTERNAL_ERROR**
  An error in algorithm implementation occurred.

- **SPARSE_STATUS_NOT_SUPPORTED**
  The requested operation is not supported.

**mkl_sparse_copy**

Creates a copy of a matrix handle.
Syntax

```c
sparse_status_t mkl_sparse_copy (const sparse_matrix_t source, const struct matrix_descr descr, sparse_matrix_t *dest);
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_copy` routine creates a copy of a matrix handle.

Input Parameters

- `source` Specifies handle containing internal data.
- `descr` Structure specifying sparse matrix properties.

- `sparse_matrix_type_t type` - Specifies the type of a sparse matrix:
  - `SPARSE_MATRIX_TYPE_GENERAL` The matrix is processed as is.
  - `SPARSE_MATRIX_TYPE_SYM MMETRIC` The matrix is symmetric (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_HERM ITIAN` The matrix is Hermitian (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_TRI ANGULAR` The matrix is triangular (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_DIAG ONAL` The matrix is diagonal (only diagonal elements are processed).
  - `SPARSE_MATRIX_TYPE_BLOCK_TRI ANGULAR` The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
  - `SPARSE_MATRIX_TYPE_BLOCK_DIA GONAL` The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

- `sparse_fill_mode_t mode` - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - `SPARSE_FILL_MODE_LOWER` The lower triangular matrix part is processed.
  - `SPARSE_FILL_MODE_UPPER` The upper triangular matrix part is processed.

- `sparse_diag_type_t diag` - Specifies diagonal type for non-general matrices:
  - `SPARSE_DIAG_NON_UNIT` Diagonal elements might not be equal to one.
  - `SPARSE_DIAG_UNIT` Diagonal elements are equal to one.
Output Parameters

\textit{dest} \hspace{1cm} Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- \textbf{SPARSE\_STATUS\_SUCCESS} \hspace{1cm} The operation was successful.
- \textbf{SPARSE\_STATUS\_NOT\_INITIALIZED} \hspace{1cm} The routine encountered an empty handle or matrix array.
- \textbf{SPARSE\_STATUS\_ALLOC\_FAILED} \hspace{1cm} Internal memory allocation failed.
- \textbf{SPARSE\_STATUS\_INVALID\_VALUE} \hspace{1cm} The input parameters contain an invalid value.
- \textbf{SPARSE\_STATUS\_EXECUTION\_FAILED} \hspace{1cm} Execution failed.
- \textbf{SPARSE\_STATUS\_INTERNAL\_ERROR} \hspace{1cm} An error in algorithm implementation occurred.
- \textbf{SPARSE\_STATUS\_NOT\_SUPPORTED} \hspace{1cm} The requested operation is not supported.

\textbf{mkl\_sparse\_destroy}

\textit{Frees memory allocated for matrix handle.}

Syntax

\begin{verbatim}
sparse_status_t mkl_sparse_destroy (sparse_matrix_t A);
\end{verbatim}

Include Files

- \textit{mkl\_spblas.h}

Description

The \textbf{mkl\_sparse\_destroy} routine frees memory allocated for matrix handle.

\textbf{NOTE}

You must free memory allocated for matrices after completing use of them. The mkl\_sparse\_destroy routine provides a utility to do so.

Input Parameters

\textit{A} \hspace{1cm} Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- \textbf{SPARSE\_STATUS\_SUCCESS} \hspace{1cm} The operation was successful.
- \textbf{SPARSE\_STATUS\_NOT\_INITIALIZED} \hspace{1cm} The routine encountered an empty handle or matrix array.
- \textbf{SPARSE\_STATUS\_ALLOC\_FAILED} \hspace{1cm} Internal memory allocation failed.
- \textbf{SPARSE\_STATUS\_INVALID\_VALUE} \hspace{1cm} The input parameters contain an invalid value.
- \textbf{SPARSE\_STATUS\_EXECUTION\_FAILED} \hspace{1cm} Execution failed.
mkl_sparse_convert_csr
Converts internal matrix representation to CSR format.

Syntax
sparse_status_t mkl_sparse_convert_csr (const sparse_matrix_t source, const sparse_operation_t operation, sparse_matrix_t *dest);

Include Files
• mkl_spblas.h

Description
The mkl_sparse_convert_csr routine converts internal matrix representation to CSR format.

Input Parameters
source
Handle containing internal data.

operation
Specifies operation \( \text{op}() \) on input matrix.

- SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, \( \text{op}(A) = A \).
- SPARSE_OPERATION_TRANSPOSE Transpose, \( \text{op}(A) = A^T \).
- SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, \( \text{op}(A) = A^H \).

Output Parameters
dest
Handle containing internal data.

Return Values
The function returns a value indicating whether the operation was successful or not, and why.

- SPARSE_STATUS_SUCCESS The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.
- SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
- SPARSE_STATUS_EXECUTION_FAILED Execution failed.
- SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
- SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
mkl_sparse_convert_bsr

Converts internal matrix representation to BSR format or changes BSR block size.

Syntax

sparse_status_t mkl_sparse_convert_bsr (const sparse_matrix_t source, const MKL_INT block_size, const sparse_layout_t block_layout, const sparse_operation_t operation, sparse_matrix_t *dest);

Include Files

• mkl_spblas.h

Description

The mkl_sparse_convert_bsr routine converts internal matrix representation to BSR format or changes BSR block size.

Input Parameters

source

Handle containing internal data.

block_size

Size of the block in the output structure.

block_layout

Specifies layout of blocks:

- SPARSE_LAYOUT_ROW_MAJOR
  - Storage of elements of blocks uses row major layout.
- SPARSE_LAYOUT_COLUMN_MAJOR
  - Storage of elements of blocks uses column major layout.

operation

Specifies operation op() on input matrix.

- SPARSE_OPERATION_NON_TRANSPOSE
  - Non-transpose, op(A) = A.
- SPARSE_OPERATION_TRANSPOSE
  - Transpose, op(A) = A^T.
- SPARSE_OPERATION_CONJ_TRANSPOSE
  - Conjugate transpose, op(A) = A^H.

Output Parameters

dest

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- SPARSE_STATUS_SUCCESS
  - The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED
  - The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_ALLOC_FAILED
  - Internal memory allocation failed.
- SPARSE_STATUS_INVALID_VALUE
  - The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED  Execution failed.
SPARSE_STATUS_INTERNAL_ERROR  An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED  The requested operation is not supported.

**mkl_sparse_?_export_csr**
Exports CSR matrix from internal representation.

**Syntax**

```c
sparse_status_t mkl_sparse_s_export_csr (const sparse_matrix_t source, sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_indx, float **values);
sparse_status_t mkl_sparse_d_export_csr (const sparse_matrix_t source, sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_indx, double **values);
sparse_status_t mkl_sparse_c_export_csr (const sparse_matrix_t source, sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_indx, MKL_Complex8 **values);
sparse_status_t mkl_sparse_z_export_csr (const sparse_matrix_t source, sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_indx, MKL_Complex16 **values);
```

**Include Files**

- mkl_spblas.h

**Description**

If the matrix specified by the `source` handle is in CSR format, the `mkl_sparse_?_export_csr` routine exports an \( m \)-by-\( k \) matrix \( A \) in CSR format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSR format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

**Input Parameters**

- `source`: Handle containing internal data.

**Output Parameters**

- `indexing`: Indicates how input arrays are indexed.
  - `SPARSE_INDEX_BASE_ZERO`: Zero-based (C-style) indexing: indices start at 0.
  - `SPARSE_INDEX_BASE_ONE`: One-based (Fortran-style) indexing: indices start at 1.

- `rows`: Number of rows of the matrix `source`.

- `cols`: Number of columns of the matrix `source`.  

288
**rows_start**

Pointer to array of length \( m \). This array contains row indices, such that \( \text{rows_start}[i] - \text{ind} \) is the first index of row \( i \) in the arrays \( \text{values} \) and \( \text{col_indx} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to \( \text{pointerB} \) array description in CSR Format for more details.

**rows_end**

Pointer to array of length \( m \). This array contains row indices, such that \( \text{rows_end}[i] - \text{ind} - 1 \) is the last index of row \( i \) in the arrays \( \text{values} \) and \( \text{col_indx} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to \( \text{pointerE} \) array description in CSR Format for more details.

**col_indx**

For one-based indexing, pointer to array containing the column indices plus one for each non-zero element of the matrix \( \text{source} \). For zero-based indexing, pointer to array containing the column indices for each non-zero element of the matrix \( \text{source} \). Its length is \( \text{rows_end}[\text{rows} - 1] - \text{ind} \). \( \text{ind} \) takes 0 for zero-based indexing and 1 for one-based indexing.

**values**

Pointer to array containing non-zero elements of the matrix \( A \). Its length is equal to length of the \( \text{col_indx} \) array.

Refer to \( \text{values} \) array description in CSR Format for more details.

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.

**mkl_sparse_?_export_csc**

Exports CSC matrix from internal representation.

**Syntax**

```c
sparse_status_t mkl_sparse_s_export_csc (const sparse_matrix_t source,
spatial_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, float **values);

sparse_status_t mkl_sparse_d_export_csc (const sparse_matrix_t source,
spatial_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, double **values);

sparse_status_t mkl_sparse_c_export_csc (const sparse_matrix_t source,
spatial_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
MKL_INT **cols_end, MKL_INT **row_indx, MKL_Complex8 **values);
```
sparse_status_t mkl_sparse_z_export_csc (const sparse_matrix_t source,
    sparse_index_base_t *indexing, MKL_INT *rows, MKL_INT *cols, MKL_INT **cols_start,
    MKL_INT **cols_end, MKL_INT **row_indx, MKL_Complex16 **values);

**Include Files**

- mkl_spblas.h

**Description**
If the matrix specified by the `source` handle is in CSC format, the `mkl_sparse_?_export_csc` routine exports an \( m \)-by-\( k \) matrix \( A \) in CSC format matrix from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in CSC format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

**Input Parameters**

- `source`  
  Handle containing internal data.

**Output Parameters**

- `indexing`  
  Indicates how input arrays are indexed.
  - `SPARSE_INDEX_BASE_ZER0`  
    Zero-based (C-style) indexing: indices start at 0.
  - `SPARSE_INDEX_BASE_ONE`  
    One-based (Fortran-style) indexing: indices start at 1.

- `rows`  
  Number of rows of the matrix `source`.

- `cols`  
  Number of columns of the matrix `source`.

- `cols_start`  
  Array of length \( m \). This array contains column indices, such that \( \text{cols}_\text{start}[i] - \text{cols}_\text{start}[0] \) is the first index of column \( i \) in the arrays `values` and `row_indx`.

  Refer to `pointerb` array description in `csc Format` for more details.

- `cols_end`  
  Pointer to array of length \( m \). This array contains row indices, such that \( \text{cols}_\text{end}[i] - \text{cols}_\text{start}[0] - 1 \) is the last index of column \( i \) in the arrays `values` and `row_indx`.

  Refer to `pointerE` array description in `csc Format` for more details.

- `row_indx`  
  For one-based indexing, pointer to array containing the row indices plus one for each non-zero element of the matrix `source`. For zero-based indexing, pointer to array containing the row indices for each non-zero element of the matrix `source`. Its length is \( \text{cols}_\text{end}[\text{cols} - 1] - \text{cols}_\text{start}[0] \).

- `values`  
  Pointer to array containing non-zero elements of the matrix \( A \). Its length is equal to length of the `row_indx` array.

  Refer to `values` array description in `csc Format` for more details.

**Return Values**
The function returns a value indicating whether the operation was successful or not, and why.
The operation was successful.
The routine encountered an empty handle or matrix array.
Internal memory allocation failed.
The input parameters contain an invalid value.
Execution failed.
An error in algorithm implementation occurred.
The requested operation is not supported.

**mkl_sparse_?_export_bsr**

*Exports BSR matrix from internal representation.*

**Syntax**

```c
sparse_status_t mkl_sparse_s_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT *cols,
MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_index, float **values);

sparse_status_t mkl_sparse_d_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT *cols,
MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_index, double **values);

sparse_status_t mkl_sparse_c_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT *cols,
MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_index, MKL_Complex8 **values);

sparse_status_t mkl_sparse_z_export_bsr (const sparse_matrix_t source,
sparse_index_base_t *indexing, sparse_layout_t *block_layout, MKL_INT *rows, MKL_INT *cols,
MKL_INT *block_size, MKL_INT **rows_start, MKL_INT **rows_end, MKL_INT **col_index, MKL_Complex16 **values);
```

**Include Files**

- mkl_spblas.h

**Description**

If the matrix specified by the `source` handle is in BSR format, the `mkl_sparse_?_export_bsr` routine exports an $m$-by-$k$ matrix $A$ in BSR format from the internal representation. The routine returns pointers to the internal representation and does not allocate additional memory.

If the matrix is not already in BSR format, the routine returns `SPARSE_STATUS_INVALID_VALUE`.

**Input Parameters**

- `source` Handle containing internal data.

**Output Parameters**

- `indexing` Indicates how input arrays are indexed.
Zero-based (C-style) indexing: indices start at 0.

One-based (Fortran-style) indexing: indices start at 1.

**block_layout**

Specifies layout of blocks:

- **SPARSE_LAYOUT_ROW_MAJOR** Storage of elements of blocks uses row major layout.
- **SPARSE_LAYOUT_COLUMN_MAJOR** Storage of elements of blocks uses column major layout.

**rows**

Number of block rows of the matrix `source`.

**cols**

Number of columns of the matrix `source`. Number of block columns of matrix `source`.

**block_size**

Size of the block in matrix `source`.

**rows_start**

Pointer to array of length `m`. This array contains row indices, such that `rows_start[i] - ind` is the first index of block row `i` in the arrays `values` and `col_ind`. `ind` takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to `pointerB` array description in CSR Format for more details.

**rows_end**

Pointer to array of length `m`. This array contains row indices, such that `rows_end[i] - ind - 1` is the last index of block row `i` in the arrays `values` and `col_ind`. `ind` takes 0 for zero-based indexing and 1 for one-based indexing.

Refer to `pointerE` array description in CSR Format for more details.

**col_ind**

For one-based indexing, pointer to array containing the column indices plus one for each non-zero blocks of the matrix `source`. For zero-based indexing, pointer to array containing the column indices for each non-zero blocks of the matrix `source`. Its length is `rows_end[m - 1] - ind[0]`. `ind` takes 0 for zero-based indexing and 1 for one-based indexing.

**values**

Pointer to array containing non-zero elements of matrix `source`. Its length is equal to length of the `col_ind` array multiplied by `block_size*block_size`.

Refer to the `values` array description in BSR Format for more details.

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS** The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED** The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED** Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE** The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED** Execution failed.
SPARSE_STATUS_INTERNAL_ERROR
An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED
The requested operation is not supported.

\textbf{mkl_sparse\_?\_set_value}

Changes a single value of matrix in internal representation.

\textbf{Syntax}

sparse_status_t mkl_sparse_s_set_value (const sparse_matrix_t A, const MKL_INT row, const MKL_INT col, const float value);

sparse_status_t mkl_sparse_d_set_value (const sparse_matrix_t A, const MKL_INT row, const MKL_INT col, const double value);

sparse_status_t mkl_sparse_c_set_value (const sparse_matrix_t A, const MKL_INT row, const MKL_INT col, const MKL_Complex8 value);

sparse_status_t mkl_sparse_z_set_value (const sparse_matrix_t A, const MKL_INT row, const MKL_INT col, const MKL_Complex16 value);

\textbf{Include Files}

- mkl_spblas.h

\textbf{Description}

Use the \texttt{mkl_sparse\_?\_set_value} routine to change a single value of a matrix in the internal Inspector-executor Sparse BLAS format. The value should already be presented in a matrix structure.

\textbf{Input Parameters}

- \texttt{A} Specifies handle containing internal data.
- \texttt{row} Indicates row of matrix in which to set value.
- \texttt{col} Indicates column of matrix in which to set value.
- \texttt{value} Indicates value

\textbf{Output Parameters}

- \texttt{A} Handle containing modified internal data.

\textbf{Return Values}

The function returns a value indicating whether the operation was successful or not, and why.

- SPARSE_STATUS_SUCCESS The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
- SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

\textbf{mkl_sparse\_?\_update_values}

Changes all or selected matrix values in internal representation.
Syntax

**NOTE**
This routine is supported for sparse matrices in BSR format only.

```c
sparse_status_t mkl_sparse_s_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT *indx, MKL_INT *indy, float *values);
sparse_status_t mkl_sparse_d_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT *indx, MKL_INT *indy, double *values);
sparse_status_t mkl_sparse_c_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT *indx, MKL_INT *indy, MKL_Complex8 *values);
sparse_status_t mkl_sparse_z_update_values (sparse_matrix_t A, MKL_INT nvalues, MKL_INT *indx, MKL_INT *indy, MKL_Complex16 *values);
```

Include Files
- mkl_spblas.h

Description

Use the `mkl_sparse_?_update_values` routine to change all or selected values of a matrix in the internal Inspector-Executor Sparse BLAS format.

The values to be updated should already be present in the matrix structure.

- To change selected values, you must provide an array values (with new values) and also the corresponding row and column indices for each value via `indx` and `indy` arrays as well as the overall number of changed elements `nvalues`.

  So that, for example, to change A(0, 0) to 1 and A(0, 1) to 2, pass the following input parameters: `nvalues = 2, indx = {0, 0}, indy = {0, 1}` and `values = {1, 2}`.

- To change all the values in the matrix, one can simply provide the `values` array, no need to supply `indx` and `indy` arrays.

Input Parameters

- **A**: Specifies handle containing internal data.
- **nvalues**: Total number of elements changed.
- **indx**: Row indices for the new values.
- **indy**: Column indices for the new values.
- **values**: New values.

Output Parameters

- **A**: Handle containing modified internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- `SPARSE_STATUS_SUCCESS`: The operation was successful.
- `SPARSE_STATUS_NOT_INITIALIZED`: The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED     Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE     The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED Execution failed.
SPARSE_STATUS_INTERNAL_ERROR   An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED    The requested operation is not supported.

mkl_sparse_order
Performs ordering of column indexes of the matrix in CSR format

Syntax
sparse_status_t mkl_sparse_order (const sparse_matrix_t csrA);

Include Files
• mkl_spblas.h

Description
Use the mkl_sparse_order routine to perform ordering of column indexes of the matrix in CSR format.

Input Parameters
csrA                  CSR data

Output Parameters
csrA                  Handle containing modified internal data.

Return Values
The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS     The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

Inspector-executor Sparse BLAS Analysis Routines

Analysis Routines and their Data Types

<table>
<thead>
<tr>
<th>Routine or Function Group</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_sparse_set_lu_smoot    Provides and estimate of the number and type of upcoming calls to LU     her_hint    smoother functionality.</td>
<td></td>
</tr>
<tr>
<td>mkl_sparse_set_mv_hint     Provides estimate of number and type of upcoming matrix-vector operations.</td>
<td></td>
</tr>
<tr>
<td>mkl_sparse_set_sv_hint     Provides estimate of number and type of upcoming triangular system solver operations.</td>
<td></td>
</tr>
<tr>
<td>Routine or Function Group</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mkl_sparse_set_mm_hint</td>
<td>Provides estimate of number and type of upcoming matrix-matrix multiplication operations.</td>
</tr>
<tr>
<td>mkl_sparse_set_sm_hint</td>
<td>Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.</td>
</tr>
<tr>
<td>mkl_sparse_set_dotmv_hint</td>
<td>Sets estimate of the number and type of upcoming matrix-vector operations.</td>
</tr>
<tr>
<td>mkl_sparse_set_sygms_hint</td>
<td>Sets estimate of number and type of upcoming mkl_sparse_?_symgs operations.</td>
</tr>
<tr>
<td>mkl_sparse_set_memory_hint</td>
<td>Provides memory requirements for performance optimization purposes.</td>
</tr>
<tr>
<td>mkl_sparse_optimize</td>
<td>Analyzes matrix structure and performs optimizations using the hints provided in the handle.</td>
</tr>
</tbody>
</table>

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**mkl_sparse_set_lu_smoother_hint**

*Provides an estimate of the number and type of upcoming calls to LU smoother functionality.*

**Syntax**

```c
sparse_status_t mkl_sparse_set_lu_smoother_hint (sparse_matrix_t A, const sparse_operation_t operation, struct matrix_descr descr, MKL_INT expected_calls);
```

**Include Files**

- mkl_spblas.h

**Description**

The `mkl_sparse_set_lu_smoother_hint` function provides subsequent Inspector-Executor Sparse BLAS calls an estimate of the number of upcoming calls to the `lu_smoother` routine that ultimately may influence the optimizations applied and specifies whether or not to perform an operation on the matrix.
Optimization Notice

Optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

Specifies the operation \( op() \) on input matrix.

- \texttt{SPARSE\_OPERATION\_NON\_TRANSPOSE}\n  Non-transpose, \( \text{op}(A) = A \).
- \texttt{SPARSE\_OPERATION\_TRANSPOSE}\n  Transpose, \( \text{op}(A) = A^T \).
- \texttt{SPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE}\n  Conjugate transpose, \( \text{op}(A) = A^H \).

Structure specifying sparse matrix properties.

- \texttt{sparse\_matrix\_type\_t type}\n  Specifies the type of a sparse matrix:
  - \texttt{SPARSE\_MATRIX\_TYPE\_GENERAL}\n    The matrix is processed as is.
  - \texttt{SPARSE\_MATRIX\_TYPE\_SYMMETRIC}\n    The matrix is symmetric (only the requested triangle is processed).
  - \texttt{SPARSE\_MATRIX\_TYPE\_HERMITIAN}\n    The matrix is Hermitian (only the requested triangle is processed).
  - \texttt{SPARSE\_MATRIX\_TYPE\_TRIANGULAR}\n    The matrix is triangular (only the requested triangle is processed).
  - \texttt{SPARSE\_MATRIX\_TYPE\_DIAGONAL}\n    The matrix is diagonal (only diagonal elements are processed).
  - \texttt{SPARSE\_MATRIX\_TYPE\_BLOCK\_TRIANGULAR}\n    The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.
  - \texttt{SPARSE\_MATRIX\_TYPE\_BLOCK\_DIAGONAL}\n    The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

- \texttt{sparse\_fill\_mode\_t mode}\n  Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - \texttt{SPARSE\_FILL\_MODE\_LOWER}\n    The lower triangular matrix part is processed.
  - \texttt{SPARSE\_FILL\_MODE\_UPPER}\n    The upper triangular matrix part is processed.

- \texttt{sparse\_diag\_type\_t diag}\n  Specifies the diagonal type for non-general matrices:
  - \texttt{SPARSE\_DIAG\_NON\_UNIT}\n    Diagonal elements might not be equal to one.
Diagonal elements are equal to one.

Number of expected calls to execution routine.

Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

**mkl_sparse_set_mv_hint**

*Provides estimate of number and type of upcoming matrix-vector operations.*

**Syntax**

```c
sparse_status_t mkl_sparse_set_mv_hint (const sparse_matrix_t A, const sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT expected_calls);
```

**Include Files**

- mkl_spblas.h

**Description**

Use the `mkl_sparse_set_mv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
**Input Parameters**

*operation*

Specifies operation $\text{op()}$ on input matrix.

- SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, $\text{op}(A) = A$.
- SPARSE_OPERATION_TRANSPOSE Transpose, $\text{op}(A) = A^T$.
- SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, $\text{op}(A) = A^H$.

*descr*

Structure specifying sparse matrix properties.

- **sparse_matrix_type_t** *type* - Specifies the type of a sparse matrix:
  - SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.
  - SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed).
  - SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed).
  - SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed).
  - SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed).
  - SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
  - SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

- **sparse_fill_mode_t** *mode* - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.
  - SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

- **sparse_diag_type_t** *diag* - Specifies diagonal type for non-general matrices:
  - SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.
  - SPARSE_DIAG_UNIT Diagonal elements are equal to one.

*expected_calls*

Number of expected calls to execution routine.

**Output Parameters**

*A*

Handle containing internal data.
Return Values
The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

**mkl_sparse_set_sv_hint**
**Provides estimate of number and type of upcoming triangular system solver operations.**

**Syntax**
```
sparse_status_t mkl_sparse_set_sv_hint (const sparse_matrix_t A, const sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT expected_calls);
```

**Include Files**
- mkl_spblas.h

**Description**
The **mkl_sparse_set_sv_hint** routine provides an estimate of the number of upcoming triangular system solver operations and type of these operations for performance optimization.

**Optimization Notice**
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

- **operation**: Specifies operation `op()` on input matrix.
  - **SPARSE_OPERATION_NON_TRANSPOSE**: Non-transpose, `op(A) = A`.
  - **SPARSE_OPERATION_TRANSPOSE**: Transpose, `op(A) = A^T`.
Conjugate transpose, \( \mathbf{op}(A) = A^H \).

Structure specifying sparse matrix properties.

- **type** - Specifies the type of a sparse matrix:
  - **SPARSE_MATRIX_TYPE_GENERAL** The matrix is processed as is.
  - **SPARSE_MATRIX_TYPE_SYMMETRIC** The matrix is symmetric (only the requested triangle is processed).
  - **SPARSE_MATRIX_TYPE_HERMITIAN** The matrix is Hermitian (only the requested triangle is processed).
  - **SPARSE_MATRIX_TYPE_TRIANGULAR** The matrix is triangular (only the requested triangle is processed).
  - **SPARSE_MATRIX_TYPE_DIAGONAL** The matrix is diagonal (only diagonal elements are processed).
  - **SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR** The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
  - **SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL** The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

- **mode** - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - **SPARSE_FILL_MODE_LOWER** The lower triangular matrix part is processed.
  - **SPARSE_FILL_MODE_UPPER** The upper triangular matrix part is processed.

- **diag** - Specifies diagonal type for non-general matrices:
  - **SPARSE_DIAG_NON_UNIT** Diagonal elements might not be equal to one.
  - **SPARSE_DIAG_UNIT** Diagonal elements are equal to one.

**Output Parameters**

- **A** Handle containing internal data.

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS** The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED** The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED  Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE  The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED  Execution failed.
SPARSE_STATUS_INTERNAL_ERROR  An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED  The requested operation is not supported.

**mkl_sparse_set_mm_hint**

Provides estimate of number and type of upcoming matrix-matrix multiplication operations.

**Syntax**

```c
sparse_status_t mkl_sparse_set_mm_hint(
    const sparse_matrix_t A,
    const sparse_operation_t operation,
    const struct matrix_descr descr,
    const sparse_layout_t layout,
    const MKL_INT dense_matrix_size,
    const MKL_INT expected_calls);
```

**Include Files**

- mkl_spblas.h

**Description**

The `mkl_sparse_set_mm_hint` routine provides an estimate of the number of upcoming matrix-matrix multiplication operations and type of these operations for performance optimization purposes.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

- **operation**
  Specifies operation \( \text{op}() \) on input matrix.
  - `SPARSE_OPERATION_NON_TRANSPOSE`  Non-transpose, \( \text{op}(A) = A \).
  - `SPARSE_OPERATION_TRANSPOSE`  Transpose, \( \text{op}(A) = A^T \).
  - `SPARSE_OPERATION_CONJUGATE_TRANSPOSE`  Conjugate transpose, \( \text{op}(A) = A^H \).

- **descr**
  Structure specifying sparse matrix properties.
  - `sparse_matrix_type_t type` - Specifies the type of a sparse matrix:
SPARSE_MATRIX_TYPE_GENERAL  The matrix is processed as is.
SPARSE_MATRIX_TYPE_SYMMETRIC  The matrix is symmetric (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_HERMITIAN  The matrix is Hermitian (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_TRIANGULAR  The matrix is triangular (only the requested triangle is processed).
SPARSE_MATRIX_TYPE_DIAGONAL  The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR  The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL  The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
SPARSE_FILL_MODE_LOWER  The lower triangular matrix part is processed.
SPARSE_FILL_MODE_UPPER  The upper triangular matrix part is processed.

sparse_diag_type_t diag - Specifies diagonal type for non-general matrices:
SPARSE_DIAG_NON_UNIT  Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT  Diagonal elements are equal to one.

layout

SPARSE_LAYOUT_COLUMN_MAJOR  Storage of elements uses column major layout.
SPARSE_LAYOUT_ROW_MAJOR  Storage of elements uses row major layout.

dense_matrix_size

Number of columns in dense matrix.

expected_calls

Number of expected calls to execution routine.

Output Parameters

A  Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.
SPARSE_STATUS_SUCCESS  The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_set_sm_hint
Provides estimate of number and type of upcoming triangular matrix solve with multiple right hand sides operations.

Syntax
sparse_status_t mkl_sparse_set_sm_hint (const sparse_matrix_t A, const sparse_operation_t operation, const struct matrix_descr descr, const sparse_layout_t layout, const MKL_INT dense_matrix_size, const MKL_INT expected_calls);

Include Files
• mkl_spblas.h

Description
The mkl_sparse_set_sm_hint routine provides an estimate of the number of upcoming triangular matrix solve with multiple right hand sides operations and type of these operations for performance optimization purposes.

Optimization Notice
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microarchitectures. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters
operation
Specifies operation op() on input matrix.

Sparse_operation_non_transpose, \( \text{op}(A) = A \)

Sparse_operation_transpose, \( \text{op}(A) = A^T \)

Sparse_operation_conjugate_transpose, \( \text{op}(A) = A^H \)

descr
Structure specifying sparse matrix properties.
sparse_matrix_type_t  type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL
The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC
The matrix is symmetric (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_HERMITIAN
The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_TRIANGULAR
The matrix is triangular (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_DIAGONAL
The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR
The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL
The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t  mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER
The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER
The upper triangular matrix part is processed.

sparse_diag_type_t  diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT
Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT
Diagonal elements are equal to one.

layout
Specifies layout of elements:

SPARSE_LAYOUT_COLUMN_MAJOR
Storage of elements uses column major layout.

SPARSE_LAYOUT_ROW_MAJOR
Storage of elements uses row major layout.

dense_matrix_size
Number of right-hand-side.

expected_calls
Number of expected calls to execution routine.

Output Parameters
A
Handle containing internal data.

Return Values
The function returns a value indicating whether the operation was successful or not, and why.
The operation was successful.

The routine encountered an empty handle or matrix array.

Internal memory allocation failed.

The input parameters contain an invalid value.

Execution failed.

An error in algorithm implementation occurred.

The requested operation is not supported.

**mkl_sparse_set_dotmv_hint**

Sets estimate of the number and type of upcoming matrix-vector operations.

**Syntax**

```c
sparse_status_t mkl_sparse_set_dotmv_hint (const sparse_matrix_t A, const
sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT
expected_calls);
```

**Include Files**

• mkl_spblas.h

**Description**

Use the `mkl_sparse_set_dotmv_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming matrix-vector multiplication operations for performance optimization, and specify whether or not to perform an operation on the matrix.

**Input Parameters**

- **operation**

  Specifies the operation performed on matrix \( A \).

  - If `operation` = SPARSE_OPERATION_NON_TRANSPOSE, \( \text{op}(A) = A \).
  - If `operation` = SPARSE_OPERATION_TRANSPOSE, \( \text{op}(A) = A^T \).
  - If `operation` = SPARSE_OPERATION_CONJUGATE_TRANSPOSE, \( \text{op}(A) = A^H \).

- **descr**

  Structure specifying sparse matrix properties.

  - `sparse_matrix_type_t type` - Specifies the type of a sparse matrix:
    - SPARSE_MATRIX_TYPE_GENERAL: The matrix is processed as is.
    - SPARSE_MATRIX_TYPE_SYMMETRIC: The matrix is symmetric (only the requested triangle is processed).
    - SPARSE_MATRIX_TYPE_HERMITIAN: The matrix is Hermitian (only the requested triangle is processed).
    - SPARSE_MATRIX_TYPE_TRIANGULAR: The matrix is triangular (only the requested triangle is processed).
    - SPARSE_MATRIX_TYPE_DIAGONAL: The matrix is diagonal (only diagonal elements are processed).
Sparse Matrix Types:

- **SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR**
  - The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

- **SPARSE_MATRIX_TYPE_BLOCK.Diagonal**
  - The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

Sparse Fill Mode:

- `sparse_fill_mode_t mode` - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - **SPARSE_FILL_MODE_LOWER**
    - The lower triangular matrix part is processed.
  - **SPARSE_FILL_MODE_UPPER**
    - The upper triangular matrix part is processed.

Sparse Diag Type:

- `sparse_diag_type_t diag` - Specifies diagonal type for non-general matrices:
  - **SPARSE_DIAG_NON_UNIT**
    - Diagonal elements might not be equal to one.
  - **SPARSE_DIAG_UNIT**
    - Diagonal elements are equal to one.

Output Parameters:

- **A**
  - Handle containing internal data.

Return Values:

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.

### mkl_sparse_set_symgs_hint

Syntax:

```c
sparse_status_t mkl_sparse_set_symgs_hint (const sparse_matrix_t A, const sparse_operation_t operation, const struct matrix_descr descr, const MKL_INT expected_calls);
```

Include Files:

- `mkl_spblas.h`
Description

Use the `mkl_sparse_set_symgs_hint` routine to provide the Inspector-executor Sparse BLAS API an estimate of the number of upcoming symmetric Gauss-Zeidel preconditioner operations for performance optimization, and specify whether or not to perform an operation on the matrix.

Input Parameters

`operation` Specifies the operation performed on matrix $A$.

- If $operation = SPARSE_OPERATION_NON_TRANSPOSE$, $\text{op}(A) = A$.
- If $operation = SPARSE_OPERATION_TRANSPOSE$, $\text{op}(A) = A^T$.
- If $operation = SPARSE_OPERATION_CONJUGATE_TRANSPOSE$, $\text{op}(A) = A^H$.

`descr` Structure specifying sparse matrix properties.

- `sparse_matrix_type_t type` - Specifies the type of a sparse matrix:
  - `SPARSE_MATRIX_TYPE_GENERAL` - The matrix is processed as is.
  - `SPARSE_MATRIX_TYPE_SYMMETRIC` - The matrix is symmetric (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_HERMITIAN` - The matrix is Hermitian (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_TRIANGULAR` - The matrix is triangular (only the requested triangle is processed).
  - `SPARSE_MATRIX_TYPE_DIAGONAL` - The matrix is diagonal (only diagonal elements are processed).
  - `SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR` - The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
  - `SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL` - The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

- `sparse_fill_mode_t mode` - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - `SPARSE_FILL_MODE_LOWER` - The lower triangular matrix part is processed.
  - `SPARSE_FILL_MODE_UPPER` - The upper triangular matrix part is processed.

- `sparse_diag_type_t diag` - Specifies diagonal type for non-general matrices:
  - `SPARSE_DIAG_NON_UNIT` - Diagonal elements might not be equal to one.
  - `SPARSE_DIAG_UNIT` - Diagonal elements are equal to one.

`diag` Specifies diagonal type for non-general matrices.
mode Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices.

type Specifies the type of a sparse matrix.

expected_calls Estimate of the number to the execution routine.

Output Parameters

A Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_STATUS_SUCCESS</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>SPARSE_STATUS_NOT_INITIALIZED</td>
<td>The routine encountered an empty handle or matrix array.</td>
</tr>
<tr>
<td>SPARSE_STATUS_ALLOC_FAILED</td>
<td>Internal memory allocation failed.</td>
</tr>
<tr>
<td>SPARSE_STATUS_INVALID_VALUE</td>
<td>The input parameters contain an invalid value.</td>
</tr>
<tr>
<td>SPARSE_STATUS_EXECUTION_FAILED</td>
<td>Execution failed.</td>
</tr>
<tr>
<td>SPARSE_STATUS_INTERNAL_ERROR</td>
<td>An error in algorithm implementation occurred.</td>
</tr>
<tr>
<td>SPARSE_STATUS_NOT_SUPPORTED</td>
<td>The requested operation is not supported.</td>
</tr>
</tbody>
</table>

mkl_sparse_set_memory_hint

Provides memory requirements for performance optimization purposes.

Syntax

```c
sparse_status_t mkl_sparse_set_memory_hint (const sparse_matrix_t A, const sparse_memory_usage_t policy);
```

Include Files

- mkl_spblas.h

Description

The mkl_sparse_set_memory_hint routine allocates additional memory for further performance optimization purposes.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Input Parameters

Specify memory utilization policy for optimization routine using these types:

- **SPARSE_MEMORY_NONE**: Routine can allocate memory only for auxiliary structures (such as for workload balancing); the amount of memory is proportional to vector size.

- **SPARSE_MEMORY_AGGRESSIVE**: Default. Routine can allocate memory up to the size of matrix $A$ for converting into the appropriate sparse format.

Output Parameters

- **A**: Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

**mkl_sparse_optimize**

Analyzes matrix structure and performs optimizations using the hints provided in the handle.

**Syntax**

```c
sparse_status_t mkl_sparse_optimize (sparse_matrix_t A);
```

**Include Files**

- `mkl_spblas.h`

**Description**

The `mkl_sparse_optimize` routine analyzes matrix structure and performs optimizations using the hints provided in the handle. Generally, specifying a higher number of expected operations allows for more aggressive and time consuming optimizations.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or
Input Parameters

A
Handle containing internal data.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

Inspector-Executor Sparse BLAS Execution Routines

Execution Routines and Their Data Types

<table>
<thead>
<tr>
<th>Routine or Function Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_sparse_?_lu_smother</td>
<td>s, d, c, z</td>
<td>Computes an action of a preconditioner which corresponds to the approximate matrix decomposition $A \approx (L+D)<em>E</em>(U+D)$ for the system $Ax = b$</td>
</tr>
<tr>
<td>mkl_sparse_?_mv</td>
<td>s, d, c, z</td>
<td>Computes a sparse matrix-vector product.</td>
</tr>
<tr>
<td>mkl_sparse_?_trsv</td>
<td>s, d, c, z</td>
<td>Solves a system of linear equations for a square sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_mm</td>
<td>s, d, c, z</td>
<td>Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_trsm</td>
<td>s, d, c, z</td>
<td>Solves a system of linear equations with multiple right-hand sides for a square sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_add</td>
<td>s, d, c, z</td>
<td>Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_spmm</td>
<td>s, d, c, z</td>
<td>Computes the product of two sparse matrices and stores the result in a newly allocated sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_spmmmd</td>
<td>s, d, c, z</td>
<td>Computes the product of two sparse matrices and stores the result as a dense matrix.</td>
</tr>
<tr>
<td>Routine or Function Group</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mkl_sparse_sp2m</td>
<td>s, d, c, z</td>
<td>Computes the product of two sparse matrices (support operations on both matrices) and stores the result in a newly allocated sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_sypr</td>
<td>s, d, c, z</td>
<td>Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_syprd</td>
<td>s, d, c, z</td>
<td>Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_symgs</td>
<td>s, d, c, z</td>
<td>Computes an action of a symmetric Gauss-Seidel preconditioner.</td>
</tr>
<tr>
<td>mkl_sparse_?_symgs_mv</td>
<td>s, d, c, z</td>
<td>Computes an action of a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication at the end.</td>
</tr>
<tr>
<td>mkl_sparse_?_syrkd</td>
<td>s, d, c, z</td>
<td>Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.</td>
</tr>
<tr>
<td>mkl_sparse_syrk</td>
<td>s, d, c, z</td>
<td>Computes the product of a sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.</td>
</tr>
<tr>
<td>mkl_sparse_?_dotmv</td>
<td>s, d, c, z</td>
<td>Computes a sparse matrix-vector product followed by a dot product.</td>
</tr>
</tbody>
</table>

### mkl_sparse_?_lu_smoother

*Computes an action of a preconditioner which corresponds to the approximate matrix decomposition*

\[ A \approx (L + D) \times E \times (U + D) \]  
*for the system*  
\[ Ax = b \]  
*(see description below).*

#### Syntax

```c
sparse_status_t mkl_sparse_s_lu_smoother (const sparse_operation_t op, const sparse_matrix_t A, const struct matrix descr descr, const float *diag, const float *approx_diag_inverse, float *x, const float *b);
```

```c
sparse_status_t mkl_sparse_d_lu_smoother (const sparse_operation_t op, const sparse_matrix_t A, const struct matrix descr descr, const double *diag, const double *approx_diag_inverse, double *x, const double *b);
```

```c
sparse_status_t mkl_sparse_c_lu_smoother (const sparse_operation_t op, const sparse_matrix_t A, const struct matrix descr descr, const MKL_COMPLEX8 *diag, const MKL_COMPLEX8 *approx_diag_inverse, MKL_COMPLEX8 *x, const MKL_COMPLEX8 *b);
```

```c
sparse_status_t mkl_sparse_z_lu_smoother (const sparse_operation_t op, const sparse_matrix_t A, const struct matrix descr descr, const MKL_COMPLEX16 *diag, const MKL_COMPLEX16 *approx_diag_inverse, MKL_COMPLEX16 *x, const MKL_COMPLEX16 *b);
```

#### Include Files

- mkl_spblas.h
Description

This routine computes an update for an iterative solution \( x \) of the system \( Ax = b \) by means of applying one iteration of an approximate preconditioner which is based on the following approximation:

\[
A - (L + D) \ast E \ast (U + D),
\]
where \( E \) is an approximate inverse of the diagonal (using exact inverse will result in Gauss-Seidel preconditioner), \( L \) and \( U \) are lower/upper triangular parts of \( A \), \( D \) is the diagonal (block diagonal in case of BSR format) of \( A \).

The \texttt{mkl_sparse_?_lu_smoother} routine performs these operations:

\[
\begin{align*}
    r &= b - A \ast x & \text{ /* 1. Computes the residual */ } \\
    (L + D) \ast E \ast (U + D) \ast dx &= r & \text{ /* 2. Finds the update } dx \text{ by solving the system */ } \\
    y &= x + dx & \text{ /* 3. Performs an update */ }
\end{align*}
\]

**NOTE**

This routine is supported only for sparse matrices in BSR format, non-transpose operation and real data types.

Both diagonal values and approximate diagonal inverse arrays should be passed explicitly.

Input Parameters

**operation**

Specifies the operation performed on matrix \( A \).

\[
\text{SPARSE\_OPERATION\_NON\_TRANSPOSE, } \text{op}(A) := A
\]

**NOTE**

Transpose and conjugate transpose (\text{SPARSE\_OPERATION\_TRANSPOSE} and \text{SPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE}) are not supported.

Non-transpose, \( \text{op}(A) = A \).

**A**

Handle which contains the sparse matrix \( A \).

**descr**

Structure specifying sparse matrix properties.

**sparse\_matrix\_type\_t type** - Specifies the type of a sparse matrix:

- \text{SPARSE\_MATRIX\_TYPE\_GENERAL} - The matrix is processed as is.

- \text{SPARSE\_MATRIX\_TYPE\_SYMMETRIC} - The matrix is symmetric (only the requested triangle is processed).

- \text{SPARSE\_MATRIX\_TYPE\_HERMITIAN} - The matrix is Hermitian (only the requested triangle is processed).

- \text{SPARSE\_MATRIX\_TYPE\_TRIANGULAR} - The matrix is triangular (only the requested triangle is processed).

- \text{SPARSE\_MATRIX\_TYPE\_DIAGONAL} - The matrix is diagonal (only diagonal elements are processed).
SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR  The matrix is block-triangular (only the requested triangle is processed). Applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGNOSTIC  The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER  The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER  The upper triangular matrix part is processed.

sparse_diag_type_t diag - Specifies the diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT  Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT  Diagonal elements are equal to one.

**NOTE**
Only SPARSE_MATRIX_TYPE_GENERAL is supported.

diag

Array of size at least \( m \), where \( m \) is the number of rows (or nrows * block_size * block_size in case of BSR format) of matrix \( A \).

The array diag must contain the diagonal values of matrix \( A \).

approx_diag_inverse

Array of size at least \( m \), where \( m \) is the number of rows (or the number of rows * block_size * block_size in case of BSR format) of matrix \( A \).

The array approx_diag_inverse will be used as \( E \), approximate inverse of the diagonal of the matrix \( A \).

x

Array of size at least \( k \), where \( k \) is the number of columns (or columns * block_size in case of BSR format) of matrix \( A \).

On entry, the array \( x \) must contain the input vector.

b

Array of size at least \( m \), where \( m \) is the number of rows (or rows * block_size in case of BSR format) of matrix \( A \). The array \( b \) must contain the values of the right-hand side of the system.

**Output Parameters**

\( x \)

Overwritten by the computed vector \( y \).

**Return Values**
The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS  The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED  The routine encountered an empty handle or matrix array.
**SPARSE_STATUS_ALLOC_FAILED**
Internal memory allocation failed.

**SPARSE_STATUS_INVALID_VALUE**
The input parameters contain an invalid value.

**SPARSE_STATUS_EXECUTION_FAILED**
Execution failed.

**SPARSE_STATUS_INTERNAL_ERROR**
An error in algorithm implementation occurred.

**SPARSE_STATUS_NOT_SUPPORTED**
The requested operation is not supported.

---

**mkl_sparse_?_mv**
*Computes a sparse matrix-vector product.*

**Syntax**

```c
sparse_status_t mkl_sparse_s_mv(
    const sparse_operation_t operation,
    const float alpha,
    const sparse_matrix_t A,
    const struct matrix_descr descr,
    const float *x,
    const float beta,
    float *y);
```

```c
sparse_status_t mkl_sparse_d_mv(
    const sparse_operation_t operation,
    const double alpha,
    const sparse_matrix_t A,
    const struct matrix_descr descr,
    const double *x,
    const double beta,
    double *y);
```

```c
sparse_status_t mkl_sparse_c_mv(
    const sparse_operation_t operation,
    const MKL_Complex8 alpha,
    const sparse_matrix_t A,
    const struct matrix_descr descr,
    const MKL_Complex8 *x,
    const MKL_Complex8 beta,
    MKL_Complex8 *y);
```

```c
sparse_status_t mkl_sparse_z_mv(
    const sparse_operation_t operation,
    const MKL_Complex16 alpha,
    const sparse_matrix_t A,
    const struct matrix_descr descr,
    const MKL_Complex16 *x,
    const MKL_Complex16 beta,
    MKL_Complex16 *y);
```

**Include Files**

- mkl_spblas.h

**Description**

The *mkl_sparse_?_mv* routine computes a sparse matrix-vector product defined as

\[ y := \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y \]

where:

- \( \alpha \) and \( \beta \) are scalars, \( x \) and \( y \) are vectors, and \( A \) is a sparse matrix handle of a matrix with \( m \) rows and \( k \) columns, and \( \text{op} \) is a matrix modifier for matrix \( A \).

**Input Parameters**

**operation**

- Specifies operation \( \text{op}() \) on input matrix.
  - `SPARSE_OPERATION_NON_TRANSPOSE` Non-transpose, \( \text{op}(A) = A \).
  - `SPARSE_OPERATION_TRANSPOSE` Transpose, \( \text{op}(A) = A^T \).
  - `SPARSE_OPERATION_CONJUGATE_TRANSPOSE` Conjugate transpose, \( \text{op}(A) = A^H \).

**alpha**

- Specifies the scalar \( \alpha \).

**A**

- Handle which contains the input matrix \( A \).
Structure specifying sparse matrix properties.

`sparse_matrix_type_t type` - Specifies the type of a sparse matrix:

- **SPARSE_MATRIX_TYPE_GENERAL**:
The matrix is processed as is.

- **SPARSE_MATRIX_TYPE_SYMMETRIC**:
The matrix is symmetric (only the requested triangle is processed).

- **SPARSE_MATRIX_TYPE_HERMITIAN**:
The matrix is Hermitian (only the requested triangle is processed).

- **SPARSE_MATRIX_TYPE_TRIANGULAR**:
The matrix is triangular (only the requested triangle is processed).

- **SPARSE_MATRIX_TYPE_DIAGONAL**:
The matrix is diagonal (only diagonal elements are processed).

- **SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR**:
The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

- **SPARSE_MATRIX_TYPE_BLOCK_DIAGNOSTIC**:
The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t mode` - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- **SPARSE_FILL_MODE_LOWER**:
The lower triangular matrix part is processed.

- **SPARSE_FILL_MODE_UPPER**:
The upper triangular matrix part is processed.

`sparse_diag_type_t diag` - Specifies diagonal type for non-general matrices:

- **SPARSE_DIAG_NON_UNIT**:
  Diagonal elements might not be equal to one.

- **SPARSE_DIAG_UNIT**:
  Diagonal elements are equal to one.

Array of size equal to the number of columns, \( k \) of \( A \) if \( operation = SPARSE_OPERATION_NON_TRANSPOSE \) and at least the number of rows, \( m \), of \( A \) otherwise. On entry, the array must contain the vector \( x \).

Specifies the scalar \( beta \).

Array with size at least \( m \) if \( operation=SPARSE_OPERATION_NON_TRANSPOSE \) and at least \( k \) otherwise. On entry, the array \( y \) must contain the vector \( y \). Array of size equal to the number of rows, \( m \) of \( A \) if \( operation = SPARSE_OPERATION_NON_TRANSPOSE \) and at least the number of columns, \( k \), of \( A \) otherwise. On entry, the array \( y \) must contain the vector \( y \).

Output Parameters

\( y \) - Overwritten by the updated vector \( y \).
Return Values
The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

**mkl_sparse_?_trsv**
Solves a system of linear equations for a triangular sparse matrix.

Syntax

```c
sparse_status_t mkl_sparse_s_trsv (const sparse_operation_t operation, const float alpha, const sparse_matrix_t A, const struct matrix_descr descr, const float *x, float *y);
```

```c
sparse_status_t mkl_sparse_d_trsv (const sparse_operation_t operation, const double alpha, const sparse_matrix_t A, const struct matrix_descr descr, const double *x, double *y);
```

```c
sparse_status_t mkl_sparse_c_trsv (const sparse_operation_t operation, const MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 *x, MKL_Complex8 *y);
```

```c
sparse_status_t mkl_sparse_z_trsv (const sparse_operation_t operation, const MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 *x, MKL_Complex16 *y);
```

**Include Files**
- mkl_spblas.h

**Description**
The **mkl_sparse_?_trsv** routine solves a system of linear equations for a matrix:

```
op(A)*y = alpha * x
```

where **A** is a triangular sparse matrix, **op** is a matrix modifier for matrix **A**, **alpha** is a scalar, and **x** and **y** are vectors.

**NOTE**
For sparse matrices in the BSR format, the supported combinations of 
(indexing,block_layout) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)
Input Parameters

**operation**

Specifies operation \( \text{op()} \) on input matrix.

- **SPARSE_OPERATION_NON_TRANSPOSE**  
  Non-transpose, \( \text{op}(A) = A \).
- **SPARSE_OPERATION_TRANSPOSE**  
  Transpose, \( \text{op}(A) = A^T \).
- **SPARSE_OPERATION_CONJ_TRANSPOSE**  
  Conjugate transpose, \( \text{op}(A) = A^H \).

**alpha**

Specifies the scalar \( \alpha \).

**A**

Handle which contains the input matrix \( A \).

**descr**

Structure specifying sparse matrix properties.

**sparse_matrix_type_t type** - Specifies the type of a sparse matrix:

- **SPARSE_MATRIX_TYPE_GENERAL**  
  The matrix is processed as is.
- **SPARSE_MATRIX_TYPE_SYMMETRIC**  
  The matrix is symmetric (only the requested triangle is processed).
- **SPARSE_MATRIX_TYPE_HERMITIAN**  
  The matrix is Hermitian (only the requested triangle is processed).
- **SPARSE_MATRIX_TYPE_TRIANGULAR**  
  The matrix is triangular (only the requested triangle is processed).
- **SPARSE_MATRIX_TYPE_DIAGONAL**  
  The matrix is diagonal (only diagonal elements are processed).
- **SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR**  
  The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
- **SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL**  
  The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

**sparse_fill_mode_t mode** - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- **SPARSE_FILL_MODE_LOWER**  
  The lower triangular matrix part is processed.
- **SPARSE_FILL_MODE_UPPER**  
  The upper triangular matrix part is processed.

**sparse_diag_type_t diag** - Specifies diagonal type for non-general matrices:

- **SPARSE_DIAG_NON_UNIT**  
  Diagonal elements might not be equal to one.
- **SPARSE_DIAG_UNIT**  
  Diagonal elements are equal to one.

**x**

Array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \). On entry, the array must contain the vector \( x \).
Output Parameters

\[ y \]  
Array of size at least \( m \) containing the solution to the system of linear equations.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- \texttt{SPARSE_STATUS_SUCCESS}  
The operation was successful.
- \texttt{SPARSE_STATUS_NOT_INITIALIZED}  
The routine encountered an empty handle or matrix array.
- \texttt{SPARSE_STATUS_ALLOC_FAILED}  
Internal memory allocation failed.
- \texttt{SPARSE_STATUS_INVALID_VALUE}  
The input parameters contain an invalid value.
- \texttt{SPARSE_STATUS_EXECUTION_FAILED}  
Execution failed.
- \texttt{SPARSE_STATUS_INTERNAL_ERROR}  
An error in algorithm implementation occurred.
- \texttt{SPARSE_STATUS_NOT_SUPPORTED}  
The requested operation is not supported.

\texttt{mkl_sparse_?_mm}

Computes the product of a sparse matrix and a dense matrix and stores the result as a dense matrix.

Syntax

\begin{verbatim}
sparse_status_t mkl_sparse_s_mm (const sparse_operation_t operation, const float alpha,  
const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout,  
const float *x, const MKL_INT columns, const MKL_INT idx, const float beta, float *y,  
const MKL_INT ldy);
sparse_status_t mkl_sparse_d_mm (const sparse_operation_t operation, const double alpha,  
const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout,  
const double *x, const MKL_INT columns, const MKL_INT idx, const double beta,  
double *y, const MKL_INT ldy);
sparse_status_t mkl_sparse_c_mm (const sparse_operation_t operation, const MKL_Complex8 alpha,  
const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout,  
const MKL_Complex8 *x, const MKL_INT columns, const MKL_INT idx, const MKL_Complex8 beta,  
MKL_Complex8 *y, const MKL_INT ldy);
sparse_status_t mkl_sparse_z_mm (const sparse_operation_t operation, const MKL_Complex16 alpha,  
const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout,  
const MKL_Complex16 *x, const MKL_INT columns, const MKL_INT idx, const MKL_Complex16 beta,  
MKL_Complex16 *y, const MKL_INT ldy);
\end{verbatim}

Include Files

- mkl_spblas.h

Description

The \texttt{mkl_sparse_?_mm} routine performs a matrix-matrix operation:

\[ Y := alpha \cdot \text{op}(A) \cdot X + beta \cdot Y \]
where \( \alpha \) and \( \beta \) are scalars, \( A \) is a sparse matrix, \( \text{op} \) is a matrix modifier for matrix \( A \), and \( X \) and \( Y \) are dense matrices.

The \texttt{mkl_sparse_?_mm} and \texttt{mkl_sparse_?_trsm} routines support these configurations:

<table>
<thead>
<tr>
<th>0-based sparse matrix:</th>
<th>Column-major dense matrix:</th>
<th>Row-major dense matrix:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_INDEX_BASE_ZERO</td>
<td>( \text{layout} = ) SPARSE_LAYOUT_COLUMN_MAJOR</td>
<td>( \text{layout} = ) SPARSE_LAYOUT_ROW_MAJOR</td>
</tr>
<tr>
<td>CSR</td>
<td>All formats</td>
<td>CSR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1-based sparse matrix:</th>
<th>CSR</th>
<th>BSR: general non-transposed matrix multiplication only</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_INDEX_BASE_ONE</td>
<td>All formats</td>
<td>All formats</td>
</tr>
</tbody>
</table>

\textbf{NOTE}
For sparse matrices in the BSR format, the supported combinations of \((\text{indexing,block\_layout})\) are:

- \((\text{SPARSE\_INDEX\_BASE\_ZERO, SPARSE\_LAYOUT\_ROW\_MAJOR})\)
- \((\text{SPARSE\_INDEX\_BASE\_ONE, SPARSE\_LAYOUT\_COLUMN\_MAJOR})\)

\textbf{Input Parameters}

- \textit{operation} Specifies operation \( \text{op}() \) on input matrix.
  
  - \texttt{SPARSE\_OPERATION\_NON\_TRANSPOSE} Non-transpose, \( \text{op}(A) = A \).
  
  - \texttt{SPARSE\_OPERATION\_TRANSPOSE} Transpose, \( \text{op}(A) = A^T \).
  
  - \texttt{SPARSE\_OPERATION\_CONJUATE\_TRANSPOSE} Conjugate transpose, \( \text{op}(A) = A^H \).

- \textit{alpha} Specifies the scalar \( \alpha \).

- \textit{A} Handle which contains the sparse matrix \( A \).

- \textit{descr} Structure specifying sparse matrix properties.
  
  - \texttt{SPARSE\_MATRIX\_TYPE\_GENERAL} The matrix is processed as is.
  
  - \texttt{SPARSE\_MATRIX\_TYPE\_SYMMETRIC} The matrix is symmetric (only the requested triangle is processed).
  
  - \texttt{SPARSE\_MATRIX\_TYPE\_HERMITIAN} The matrix is Hermitian (only the requested triangle is processed).
  
  - \texttt{SPARSE\_MATRIX\_TYPE\_TRIANGULAR} The matrix is triangular (only the requested triangle is processed).
The matrix is diagonal (only diagonal elements are processed).

The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

**sparse_fill_mode_t**

- **mode** - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
  - **SPARSE_FILL_MODE_LOWER** - The lower triangular matrix part is processed.
  - **SPARSE_FILL_MODE_UPPER** - The upper triangular matrix part is processed.

**sparse_diag_type_t**

- **diag** - Specifies diagonal type for non-general matrices:
  - **SPARSE_DIAG_NON_UNIT** - Diagonal elements might not be equal to one.
  - **SPARSE_DIAG_UNIT** - Diagonal elements are equal to one.

**layout**

Describes the storage scheme for the dense matrix:

- **SPARSE_LAYOUT_COLUMN_MAJOR** - Storage of elements uses column major layout.
- **SPARSE_LAYOUT_ROW_MAJOR** - Storage of elements uses row major layout.

**x**

Array of size at least rows*cols.

<table>
<thead>
<tr>
<th>rows (number of rows in x)</th>
<th>ldx</th>
</tr>
</thead>
</table>

If op(A) = A, number of columns in A.

If op(A) = Aᵀ, number of rows in A.

<table>
<thead>
<tr>
<th>columns (number of columns in x)</th>
<th>columns</th>
<th>ldx</th>
</tr>
</thead>
</table>

Number of columns of matrix Y.

Specifies the leading dimension of matrix X.

**beta**

Specifies the scalar beta

**y**

Array of size at least rows*cols, where
Output Parameters

\( y \)

Overwritten by the updated matrix \( Y \).

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.

**mkl_sparse_?_trsm**

Solves a system of linear equations with multiple right hand sides for a triangular sparse matrix.

**Syntax**

```c
sparse_status_t mkl_sparse_s_trsm (const sparse_operation_t operation, const float alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout, const float *x, const MKL_INT columns, const MKL_INT ldx, float *y, const MKL_INT ldy);

sparse_status_t mkl_sparse_d_trsm (const sparse_operation_t operation, const double alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout, const double *x, const MKL_INT columns, const MKL_INT ldx, double *y, const MKL_INT ldy);

sparse_status_t mkl_sparse_c_trsm (const sparse_operation_t operation, const MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout, const MKL_Complex8 *x, const MKL_INT columns, const MKL_INT ldx, MKL_Complex8 *y, const MKL_INT ldy);
```
sparse_status_t mkl_sparse_z_trsm (const sparse_operation_t operation, const MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const sparse_layout_t layout, const MKL_Complex16 *x, const MKL_INT columns, const MKL_INT ldx, MKL_Complex16 *y, const MKL_INT ldy);

Include Files

- mkl_spblas.h

Description

The mkl_sparse_?_trsm routine solves a system of linear equations with multiple right hand sides for a triangular sparse matrix:

\[ Y := alpha \cdot \text{inv}(\text{op}(A)) \cdot X \]

where:

- \( alpha \) is a scalar,
- \( X \) and \( Y \) are dense matrices,
- \( A \) is a sparse matrix, and
- \( \text{op} \) is a matrix modifier for matrix \( A \).

The mkl_sparse_?_mm and mkl_sparse_?_trsm routines support these configurations:

<table>
<thead>
<tr>
<th>Column-major dense matrix:</th>
<th>Row-major dense matrix:</th>
<th>0-based sparse matrix:</th>
<th>1-based sparse matrix:</th>
</tr>
</thead>
<tbody>
<tr>
<td>layout = SPARSE_LAYOUT_COLUMN_MAJOR</td>
<td>layout = SPARSE_LAYOUT_ROW_MAJOR</td>
<td>CSR</td>
<td>CSR</td>
</tr>
<tr>
<td>CSR: general non-transposed matrix multiplication only</td>
<td>All formats</td>
<td>All formats</td>
<td>BSR: general non-transposed matrix multiplication only</td>
</tr>
</tbody>
</table>

**NOTE**

For sparse matrices in the BSR format, the supported combinations of (indexing,block_layout) are:

- (SPARSE_INDEX_BASE_ZERO, SPARSE_LAYOUT_ROW_MAJOR)
- (SPARSE_INDEX_BASE_ONE, SPARSE_LAYOUT_COLUMN_MAJOR)

Input Parameters

**operation**

Specifies operation \( \text{op()} \) on input matrix.

- SPARSE_OPERATION_NON_TRANSPOSE Non-transpose, \( \text{op}(A) = A \).
- SPARSE_OPERATION_TRANSPOSE Transpose, \( \text{op}(A) = A^T \).
- SPARSE_OPERATION_CONJUGATE_TRANSPOSE Conjugate transpose, \( \text{op}(A) = A^H \).

**alpha**

Specifies the scalar \( alpha \).

**A**

Handle which contains the sparse matrix \( A \).
Structure specifying sparse matrix properties.

`sparse_matrix_type_t type` - Specifies the type of a sparse matrix:

- `SPARSE_MATRIX_TYPE_GENERAL` - The matrix is processed as is.
- `SPARSE_MATRIX_TYPE_SYMMETRIC` - The matrix is symmetric (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_HERMITIAN` - The matrix is Hermitian (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_TRIANGULAR` - The matrix is triangular (only the requested triangle is processed).
- `SPARSE_MATRIX_TYPE_DIAGONAL` - The matrix is diagonal (only diagonal elements are processed).
- `SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR` - The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
- `SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL` - The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

`sparse_fill_mode_t mode` - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

- `SPARSE_FILL_MODE_LOWER` - The lower triangular matrix part is processed.
- `SPARSE_FILL_MODE_UPPER` - The upper triangular matrix part is processed.

`sparse_diag_type_t diag` - Specifies diagonal type for non-general matrices:

- `SPARSE_DIAG_NON_UNIT` - Diagonal elements might not be equal to one.
- `SPARSE_DIAG_UNIT` - Diagonal elements are equal to one.

Describes the storage scheme for the dense matrix:

- `SPARSE_LAYOUT_COLUMN_MAJOR` - Storage of elements uses column major layout.
- `SPARSE_LAYOUT_ROW_MAJOR` - Storage of elements uses row major layout.

Array of size at least `rows*cols`.

<table>
<thead>
<tr>
<th>layout</th>
<th>rows (number of rows in x)</th>
<th>idx</th>
<th>number of rows in A</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SPARSE_LAYOUT_COLUMN_MAJOR</code></td>
<td><code>rows</code></td>
<td><code>idx</code></td>
<td><code>number of rows in A</code></td>
</tr>
<tr>
<td><code>SPARSE_LAYOUT_ROW_MAJOR</code></td>
<td><code>rows</code></td>
<td><code>idx</code></td>
<td><code>number of rows in A</code></td>
</tr>
</tbody>
</table>
On entry, the array \( x \) must contain the matrix \( X \).

**Number of columns in matrix \( Y \).**

**Specifies the leading dimension of matrix \( X \).**

**Array of size at least rows*cols, where**

<table>
<thead>
<tr>
<th>cols (number of columns in ( x ))</th>
<th>columns</th>
<th>ldx</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

**\( y \)**

Overwritten by the updated matrix \( Y \).

**Return Values**

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**
  - Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**
  - Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.

**mkl_sparse_?_add**

*Computes the sum of two sparse matrices. The result is stored in a newly allocated sparse matrix.*

**Syntax**

```c
sparse_status_t mkl_sparse_s_add (const sparse_operation_t operation, const sparse_matrix_t A, const float alpha, const sparse_matrix_t B, sparse_matrix_t *C);
sparse_status_t mkl_sparse_d_add (const sparse_operation_t operation, const sparse_matrix_t A, const double alpha, const sparse_matrix_t B, sparse_matrix_t *C);
sparse_status_t mkl_sparse_c_add (const sparse_operation_t operation, const sparse_matrix_t A, const MKL_Complex8 alpha, const sparse_matrix_t B, sparse_matrix_t *C);
```
sparse_status_t mkl_sparse_z_add (const sparse_operation_t operation, const sparse_matrix_t A, const MKL_Complex16 alpha, const sparse_matrix_t B, sparse_matrix_t *C);

Include Files

- mkl_spblas.h

Description

The mkl_sparse_?_add routine performs a matrix-matrix operation:

\[ C := \alpha \cdot \text{op}(A) + B \]

where \( \alpha \) is a scalar, \( \text{op} \) is a matrix modifier, and \( A, B, \) and \( C \) are sparse matrices.

NOTE

This routine is only supported for sparse matrices in CSR and BSR formats. It is not supported for COO or CSC formats.

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Handle which contains the sparse matrix ( A ).</td>
</tr>
<tr>
<td>alpha</td>
<td>Specifies the scalar ( \alpha ).</td>
</tr>
<tr>
<td>operation</td>
<td>Specifies operation ( \text{op}() ) on input matrix.</td>
</tr>
<tr>
<td></td>
<td>- ( \text{SPARSE_OPERATION_NON_TRANSPOSE} ): Non-transpose, ( \text{op}(A) = A ).</td>
</tr>
<tr>
<td></td>
<td>- ( \text{SPARSE_OPERATION_TRANPOSE} ): Transpose, ( \text{op}(A) = A^T ).</td>
</tr>
<tr>
<td></td>
<td>- ( \text{SPARSE_OPERATION_CONJUGATE_TRANSPOSE} ): Conjugate transpose, ( \text{op}(A) = A^H ).</td>
</tr>
<tr>
<td>B</td>
<td>Handle which contains the sparse matrix ( B ).</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Handle which contains the resulting sparse matrix.</td>
</tr>
</tbody>
</table>

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{SPARSE_STATUS_SUCCESS} )</td>
<td>The operation was successful.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_NOT_INITIALIZED} )</td>
<td>The routine encountered an empty handle or matrix array.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_ALLOC_FAILED} )</td>
<td>Internal memory allocation failed.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_INVALID_VALUE} )</td>
<td>The input parameters contain an invalid value.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_EXECUTION_FAILED} )</td>
<td>Execution failed.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_INTERNAL_ERROR} )</td>
<td>An error in algorithm implementation occurred.</td>
</tr>
<tr>
<td>( \text{SPARSE_STATUS_NOT_SUPPORTED} )</td>
<td>The requested operation is not supported.</td>
</tr>
</tbody>
</table>
mkl_sparse_spmm

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

sparse_status_t mkl_sparse_spmm (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, sparse_matrix_t *C);

Include Files

• mkl_spblas.h

Description

The mkl_sparse_spmm routine performs a matrix-matrix operation:

\[ C := \text{op}(A) \times B \]

where \( A \), \( B \), and \( C \) are sparse matrices and \( \text{op} \) is a matrix modifier for matrix \( A \).

NOTE

This routine is only supported for sparse matrices in CSC, CSR, and BSR formats. It is not supported for sparse matrices in COO format.

Input Parameters

- \( operation \): Specifies operation \( \text{op}() \) on input matrix.
  - \text{SPARSE_OPERATION_NON_TRANSPOSE}: Non-transpose, \( \text{op}(A) = A \).
  - \text{SPARSE_OPERATION_TRANSPOSE}: Transpose, \( \text{op}(A) = A^T \).
  - \text{SPARSE_OPERATION_CONJ_TRANSPOSE}: Conjugate transpose, \( \text{op}(A) = A^H \).

- \( A \): Handle which contains the sparse matrix \( A \).

- \( B \): Handle which contains the sparse matrix \( B \).

Output Parameters

- \( C \): Handle which contains the resulting sparse matrix.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- \text{SPARSE_STATUS_SUCCESS}: The operation was successful.
- \text{SPARSE_STATUS_NOT_INITIALIZED}: The routine encountered an empty handle or matrix array.
- \text{SPARSE_STATUS_ALLOC_FAILED}: Internal memory allocation failed.
- \text{SPARSE_STATUS_INVALID_VALUE}: The input parameters contain an invalid value.
- \text{SPARSE_STATUS_EXECUTION_FAILED}: Execution failed.
mkl_sparse_?_spmm

**Computes the product of two sparse matrices and stores the result as a dense matrix.**

**Syntax**

```c
sparse_status_t mkl_sparse_s_spmm (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, float *C, const MKL_INT ldc);

sparse_status_t mkl_sparse_d_spmm (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, double *C, const MKL_INT ldc);

sparse_status_t mkl_sparse_c_spmm (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, MKL_Complex8 *C, const MKL_INT ldc);

sparse_status_t mkl_sparse_z_spmm (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, const sparse_layout_t layout, MKL_Complex16 *C, const MKL_INT ldc);
```

**Include Files**

- mkl_spblas.h

**Description**

The `mkl_sparse_?_spmm` routine performs a matrix-matrix operation:

\[ C := \text{op}(A) \times B \]

where A and B are sparse matrices, \( \text{op} \) is a matrix modifier for matrix A, and C is a dense matrix.

**NOTE**

This routine is not supported for sparse matrices in the COO format. For sparse matrices in BSR format, these combinations of (indexing, block_layout) are supported:

- \((\text{SPARSE_INDEX_BASE_ZERO}, \text{SPARSE_LAYOUT_ROW_MAJOR})\)
- \((\text{SPARSE_INDEX_BASE_ONE}, \text{SPARSE_LAYOUT_COLUMN_MAJOR})\)

**Input Parameters**

- **operation**
  - Specifies operation \( \text{op}(\cdot) \) on input matrix.
  - `SPARSE_OPERATION_NON_TRANSPOSE` Non-transpose, \( \text{op}(A) = A \).
  - `SPARSE_OPERATION_TRANSPOSE` Transpose, \( \text{op}(A) = A^T \).
  - `SPARSE_OPERATION_CONJUGATE_TRANSPOSE` Conjugate transpose, \( \text{op}(A) = A^H \).
A Handle which contains the sparse matrix $A$.

B Handle which contains the sparse matrix $B$.

layout Describes the storage scheme for the dense matrix:

SPARSE_LAYOUT_COLUMN_MAJOR Storage of elements uses column major layout.

SPARSE_LAYOUT_ROW_MAJOR Storage of elements uses row major layout.

ldC Leading dimension of matrix $C$.

Output Parameters

$C$ Resulting dense matrix.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC FAILED Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION FAILED Execution failed.

SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.

mkl_sparse_sp2m

Computes the product of two sparse matrices. The result is stored in a newly allocated sparse matrix.

Syntax

sparse_status_t mkl_sparse_sp2m (const sparse_operation_t transA, const struct matrix_descr descrA, const sparse_matrix_t A, const sparse_operation_t transB, const struct matrix_descr descrB, const sparse_matrix_t B, const sparse_request_t request, sparse_matrix_t *C);

Include Files

• mkl_spblas.h

Description

The mkl_sparse_sp2m routine performs a matrix-matrix operation:

$$ C := \text{op}(A) \ast \text{op}(B) $$

where $A$, $B$, and $C$ are sparse matrices, $\text{op}(A)$ and $\text{op}(B)$ are matrix modifiers for matrices $A$ and $B$, respectively.
### Input Parameters

**opA**

Specifies operation on input matrix.

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_OPERATION_NON_TRANSPOSE</td>
<td>Non-transpose, ( \text{op}(A)=A )</td>
</tr>
<tr>
<td>SPARSE_OPERATION_TRANSPOSE</td>
<td>Transpose, ( \text{op}(A)=A^T )</td>
</tr>
<tr>
<td>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</td>
<td>Conjugate transpose, ( \text{op}(A)=A^H )</td>
</tr>
</tbody>
</table>

**opB**

Specifies operation on input matrix.

<table>
<thead>
<tr>
<th>Operation Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_OPERATION_NON_TRANSPOSE</td>
<td>Non-transpose, ( \text{op}(B)=B )</td>
</tr>
<tr>
<td>SPARSE_OPERATION_TRANSPOSE</td>
<td>Transpose, ( \text{op}(B)=B^T )</td>
</tr>
<tr>
<td>SPARSE_OPERATION_CONJUGATE_TRANSPOSE</td>
<td>Conjugate transpose, ( \text{op}(B)=B^H )</td>
</tr>
</tbody>
</table>

**descrA**

Structure that specifies sparse matrix properties.

**NOTE** Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

**sparse_matrix_type_t type** specifies the type of sparse matrix.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_MATRIX_TYPE_GENERAL</td>
<td>The matrix is processed as is.</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_SYMMETRIC</td>
<td>The matrix is symmetric (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_HERMITIAN</td>
<td>The matrix is Hermitian (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_TRIANGULAR</td>
<td>The matrix is triangular (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_DIAGONAL</td>
<td>The matrix is diagonal (only diagonal elements are processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</td>
<td>The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</td>
<td>The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.</td>
</tr>
</tbody>
</table>

**sparse_fill_mode_t mode** specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_FILL_MODE_LOWER</td>
<td>The lower triangular matrix is processed.</td>
</tr>
</tbody>
</table>
The upper triangular matrix is processed.

**sparse_diag_type_t** `diag` specifies the type of diagonal for non-general matrices.

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_DIAG_NON_UNIT</td>
<td>Diagonal elements must not be equal to 1.</td>
</tr>
<tr>
<td>SPARSE_DIAG_UNIT</td>
<td>Diagonal elements are equal to 1.</td>
</tr>
</tbody>
</table>

Structure that specifies sparse matrix properties.

**NOTE** Currently, only SPARSE_MATRIX_TYPE_GENERAL is supported.

**sparse_matrix_type_t** `type` specifies the type of sparse matrix.

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_MATRIX_TYPE_GENERAL</td>
<td>The matrix is processed as is.</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_SYMMETRIC</td>
<td>The matrix is symmetric (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_HERMITIAN</td>
<td>The matrix is Hermitian (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_TRIANGULAR</td>
<td>The matrix is triangular (only the requested triangle is processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_DIAGONAL</td>
<td>The matrix is diagonal (only diagonal elements are processed).</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR</td>
<td>The matrix is block-triangular (only the requested triangle is processed). This applies to BSR format only.</td>
</tr>
<tr>
<td>SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL</td>
<td>The matrix is block-diagonal (only the requested triangle is processed). This applies to BSR format only.</td>
</tr>
</tbody>
</table>

**sparse_fill_mode_t** `mode` specifies the triangular matrix portion for symmetric, Hermitian, triangular, and block-triangular matrices.

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_FILL_MODE_LOWER</td>
<td>The lower triangular matrix is processed.</td>
</tr>
<tr>
<td>SPARSE_FILL_MODE_UPPER</td>
<td>The upper triangular matrix is processed.</td>
</tr>
</tbody>
</table>

**sparse_diag_type_t** `diag` specifies the type of diagonal for non-general matrices.

<table>
<thead>
<tr>
<th>Enum</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_DIAG_NON_UNIT</td>
<td>Diagonal elements must not be equal to 1.</td>
</tr>
</tbody>
</table>
**SPARSE_DIAG_UNIT**

Diagonal elements are equal to 1.

Handle which contains the sparse matrix \( A \).

Handle which contains the sparse matrix \( B \).

Specifies whether the full computations are performed at once or using the two-stage algorithm. See Two-stage Algorithm for Inspector-executor Sparse BLAS Routines.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_STAGE_NNZ_COUNT</td>
<td>Only rowIndex (BSR/CSR format) or colIndex (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation.</td>
</tr>
<tr>
<td>SPARSE_STAGE_FINALIZE_MULT_NO_VAL</td>
<td>Finalize computations of the matrix structure (values will not be computed). Use only after the call with SPARSE_STAGE_NNZ_COUNT parameter.</td>
</tr>
<tr>
<td>SPARSE_STAGE_FINALIZE_MULT</td>
<td>Finalize computation. Can also be used when the matrix structure remains unchanged and only values of the resulting matrix ( C ) need to be recomputed.</td>
</tr>
<tr>
<td>SPARSE_STAGE_FULL_MULT_NO_VAL</td>
<td>Perform computations of the matrix structure.</td>
</tr>
<tr>
<td>SPARSE_STAGE_FULL_MULT</td>
<td>Perform the entire computation in a single step.</td>
</tr>
</tbody>
</table>

**Output Parameters**

\( C \)

Handle which contains the resulting sparse matrix.

**Return Values**

The function returns a value indicating whether the operation was successful, or the reason why it failed.

- **SPARSE_STATUS_SUCCESS**
  - The operation was successful.

- **SPARSE_STATUS_NOT_INITIALIZED**
  - The routine encountered an empty handle or matrix array.

- **SPARSE_STATUS_ALLOC_FAILED**
  - The internal memory allocation failed.

- **SPARSE_STATUS_INVALID_VALUE**
  - The input parameters contain an invalid value.

- **SPARSE_STATUS_EXECUTION_FAILED**
  - The execution failed.

- **SPARSE_STATUS_INTERNAL_ERROR**
  - An error occurred in the implementation of the algorithm.

- **SPARSE_STATUS_NOT_SUPPORTED**
  - The requested operation is not supported.
mkl_sparse_sypr

Computes the symmetric product of three sparse matrices and stores the result in a newly allocated sparse matrix.

Syntax

```c
sparse_status_t mkl_sparse_sypr (const sparse_operation_t operation, const sparse_matrix_t A, const sparse_matrix_t B, const struct matrix_descr B, sparse_matrix_t *C, const sparse_request_t request);
```

Include Files

- mkl_spblas.h

Description

The `mkl_sparse_sypr` routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, `C`.

\[ C = A \times B \times \text{op}(A) \]

or

\[ C = \text{op}(A) \times B \times A \]

depending on the matrix modifier `operation`.

Here, `A`, `B`, and `C` are sparse matrices, where `A` has a general structure while `B` and `C` are symmetric or Hermitian matrices. \text{op}A is the transpose (real precision) or conjugate transpose (complex precision) operator.

**NOTE**

This routine is not supported for sparse matrices in COO or CSC formats. This routine supports only CSR and BSR formats. In addition, it supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the `mkl_sparse_order` routine before either `mkl_sparse_sypr` or `mkl_sparse_\*syprd`.

Input Parameters

<table>
<thead>
<tr>
<th>operation</th>
<th>Specifies operation on the input sparse matrices.</th>
</tr>
</thead>
</table>
| SPARSE_OPERATION_NON_TRANSPOSE | Non-transpose case.  
\[ C = A \times B \times (A^T) \] for real precision  
\[ C = A \times B \times (A^H) \] for complex precision. |
| SPARSE_OPERATION_TRANSPOSE | Transpose case. This is not supported for complex matrices.  
\[ C = (A^T) \times B \times A \] |
SPARSE_OPERATION_CONJUGATE_TRANSPOSE

Conjugate transpose case. This is not supported for real matrices.
\[
C = (A^H)^*B*A
\]

A
Handle which contains the sparse matrix A.

B
Handle which contains the sparse matrix B.

descrB
Structure specifying properties of the sparse matrix.

sparse_matrix_type_t type specifies the type of a sparse matrix

<table>
<thead>
<tr>
<th>SPARSE_MATRIX_TYPE_SYMMETRIC</th>
<th>The matrix is symmetric (only the specified triangle is processed).</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_MATRIX_TYPE_HERMITIAN</td>
<td>The matrix is Hermitian (only the specified triangle is processed).</td>
</tr>
</tbody>
</table>

sparse_fill_mode_t mode specifies the triangular matrix part.

| SPARSE_FILL_MODE_LOWER       | The lower triangular matrix part is processed.                   |
| SPARSE_FILL_MODE_UPPER      | The upper triangular matrix part is processed.                   |

sparse_diag_type_t diag specifies the type of diagonal.

| SPARSE_DIAG_NON_UNIT        | Diagonal elements cannot be equal to one.                        |

NOTE
This routine also supports \( C = AA^T, H \) with these parameters:

descrB.type=SPARSE_MATRIX_TYPE_DIAGONAL
descrB.diag=SPARSE_DIAG_UNIT

In this case, you do not need to allocate structure B. Use the routine as a 2-stage version of mkl_sparse_syrk.

request
Use this routine to specify if the computations should be performed in a single step or using the two-stage algorithm. See Two-stage Algorithm for Inspector-executor Sparse BLAS Routines for more information.

| SPARSE_STAGE_NNZ_COUNT | Only rowIndex (BSR/CSR format) or colIndex (CSC format) array of the matrix is computed internally. The computation can be extracted to measure the memory required for full operation. |
### Output Parameters

**C**

Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.

### Return Values

The function returns a value indicating whether the operation was successful, or the reason why it failed.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

### mkl_sparse_?_syprd

*Computes the symmetric triple product of a sparse matrix and a dense matrix and stores the result as a dense matrix.*

#### Syntax

```c
sparse_status_t mkl_sparse_s_syprd (const sparse_operation_t op, const sparse_matrix_t A, const float *B, const sparse_layout_t layoutB, const MKL_INT ldb, const float alpha, const float *C, const sparse_layout_t layoutC, const MKL_INT ldc);
```

```c
sparse_status_t mkl_sparse_d_syprd (const sparse_operation_t op, const sparse_matrix_t A, const double *B, const sparse_layout_t layoutB, const MKL_INT ldb, const double alpha, const double beta, double *C, const sparse_layout_t layoutC, const MKL_INT ldc);
```
sparse_status_t mkl_sparse_c_syprd (const sparse_operation_t op, const sparse_matrix_t A, const MKL_Complex8 *B, const sparse_layout_t layoutB, const MKL_INT ldb, const MKL_Complex8 alpha, const MKL_Complex8 *C, const sparse_layout_t layoutC, const MKL_INT ldc);

sparse_status_t mkl_sparse_z_syprd (const sparse_operation_t op, const sparse_matrix_t A, const MKL_Complex16 *B, const sparse_layout_t layoutB, const MKL_INT ldb, const MKL_Complex16 alpha, const MKL_Complex16 *C, const sparse_layout_t layoutC, const MKL_INT ldc);

Include Files
• mkl_spblas.h

Description
The mkl_sparse_?_syprd routine performs a multiplication of three sparse matrices that results in a symmetric or Hermitian matrix, C.

\[ C = \alpha A B \text{op}(A) + \beta C \] or

\[ C = \alpha \text{op}(A) B A + \beta C \]

depending on the matrix modifier operation. Here A is a sparse matrix, B and C are dense and symmetric (or Hermitian) matrices.

op is the transpose (real precision) or conjugate transpose (complex precision) operator.

NOTE
This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. In addition, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If the data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_sypr or mkl_sparse_?_syprd.

Input Parameters

operation

Specifies operation on the input sparse matrix.

<table>
<thead>
<tr>
<th>Sparse Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_OPERATION_NON_TRANSPOSE</td>
<td>Non-transpose case. [ C = \alpha A B \text{op}(A) + \beta C ] for real precision. [ C = \alpha \text{op}(A) B A + \beta C ] for complex precision.</td>
</tr>
<tr>
<td>SPARSE_OPERATION_TRANSPOSE</td>
<td>Transpose case. This is not supported for complex matrices. [ C = \alpha (A^T) B A + \beta C ]</td>
</tr>
</tbody>
</table>
**SPARSE_OPERATION_CONJUGATE_TRANSPOSE**

Conjugate transpose case. This is not supported for real matrices.

\[ C := \alpha (A^H) B^* A + \beta C \]

**A**

Handle which contains the sparse matrix \( A \).

**B**

Input dense matrix. Only the upper triangular part of the matrix is used for computation.

**denselayoutB**

Structure that describes the storage scheme for the dense matrix.

<table>
<thead>
<tr>
<th>SPARSE_LAYOUT_COLUMN_MAJOR</th>
<th>Store elements in a column-major layout.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
<td>Store elements in a row-major layout.</td>
</tr>
</tbody>
</table>

**ldb**

Leading dimension of matrix \( B \).

**alpha**

Scalar parameter.

**beta**

Scalar parameter.

**NOTE**

Since the upper triangular part of matrix \( C \) is the only portion that is processed, set real values of \( \alpha \) and \( \beta \) in the complex case to obtain the Hermitian matrix.

**denselayoutC**

Structure that describes the storage scheme for the dense matrix.

<table>
<thead>
<tr>
<th>SPARSE_LAYOUT_COLUMN_MAJOR</th>
<th>Store elements in a column-major layout.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARSE_LAYOUT_ROW_MAJOR</td>
<td>Store elements in a row-major layout.</td>
</tr>
</tbody>
</table>

**ldc**

Leading dimension of matrix \( C \).

**Output Parameters**

**C**

Handle which contains the resulting dense matrix. Only the upper-triangular part of the matrix is computed.

**Return Values**

The function returns a value indicating whether the operation was successful, or the reason why it failed.

- **SPARSE_STATUS_SUCCESS**
  The operation was successful.

- **SPARSE_STATUS_NOT_INITIALIZED**
  The routine encountered an empty handle or matrix array.

- **SPARSE_STATUS_ALLOC_FAILED**
  The internal memory allocation failed.
The input parameters contain an invalid value.
The execution failed.
An error occurred in the implementation of the algorithm.
The requested operation is not supported.

**mkl_sparse_?_symgs**

*Computes a symmetric Gauss-Seidel preconditioner.*

**Syntax**

```c
sparse_status_t mkl_sparse_s_symgs (const sparse_operation_t operation, const sparse_matrix_t A, const struct matrix_descr descr, const float alpha, const float *b, float *x);
sparse_status_t mkl_sparse_d_symgs (const sparse_operation_t operation, const sparse_matrix_t A, const struct matrix_descr descr, const double alpha, const double *b, double *x);
sparse_status_t mkl_sparse_c_symgs (const sparse_operation_t operation, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 alpha, const MKL_Complex8 *b, MKL_Complex8 *x);
sparse_status_t mkl_sparse_z_symgs (const sparse_operation_t operation, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 alpha, const MKL_Complex16 *b, MKL_Complex16 *x);
```

**Include Files**

- mkl_spblas.h

**Description**

The **mkl_sparse_?_symgs** routine performs this operation:

\[
x_0 := x \cdot \alpha;
(L + D) \cdot x_1 = b - U \cdot x_0;
(U + D) \cdot x = b - L \cdot x_1;
\]

where \( A = L + D + U. \)

**NOTE**

This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the desc.type must be SPARSE_MATRIX_TYPE_SYMMETRIC.

**Input Parameters**

- **operation**
  Specifies the operation performed on matrix \( A. \)

  \( \text{SPARSE OPERATION NON_TRANSPOSE}, \) \( \text{op}(A) := A. \)
NOTE

Transpose (SPARSE_OPERATION_TRANSPOSE) and conjugate transpose (SPARSE_OPERATION_CONJUGATE_TRANSPOSE) are not supported.

A
alpha
descr

Handle which contains the sparse matrix $A$.

Specifies the scalar $alpha$.

Structure specifying sparse matrix properties.

```c
sppr`Matrix_type_t type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL
The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC
The matrix is symmetric (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_HERMITIAN
The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_TRIANGULAR
The matrix is triangular (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_DIAGONAL
The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR
The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL
The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sppr_fill_mode_t mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER
The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER
The upper triangular matrix part is processed.

sppr_diag_type_t diag - Specifies diagonal type for non-general matrices:

SPARSE_DIAG_NON_UNIT
Diagonal elements might not be equal to one.

SPARSE_DIAG_UNIT
Diagonal elements are equal to one.
```

x

Array of size at least $m$, where $m$ is the number of rows of matrix $A$.

On entry, the array $x$ must contain the vector $x$.

b

Array of size at least $m$, where $m$ is the number of rows of matrix $A$.

On entry, the array $b$ must contain the vector $b$. 
Output Parameters

- **x**: Overwritten by the computed vector x.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- **SPARSE_STATUS_SUCCESS**: The operation was successful.
- **SPARSE_STATUS_NOT_INITIALIZED**: The routine encountered an empty handle or matrix array.
- **SPARSE_STATUS_ALLOC_FAILED**: Internal memory allocation failed.
- **SPARSE_STATUS_INVALID_VALUE**: The input parameters contain an invalid value.
- **SPARSE_STATUS_EXECUTION_FAILED**: Execution failed.
- **SPARSE_STATUS_INTERNAL_ERROR**: An error in algorithm implementation occurred.
- **SPARSE_STATUS_NOT_SUPPORTED**: The requested operation is not supported.

### mkl_sparse_?_symgs_mv

*Computes a symmetric Gauss-Seidel preconditioner followed by a matrix-vector multiplication.*

#### Syntax

```c
sparse_status_t mkl_sparse_s_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const float alpha, const float *b, float *x, float *y);
```

```c
sparse_status_t mkl_sparse_d_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const double alpha, const double *b, double *x, double *y);
```

```c
sparse_status_t mkl_sparse_c_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 alpha, const
MKL_Complex8 *b, MKL_Complex8 *x, MKL_Complex8 *y);
```

```c
sparse_status_t mkl_sparse_z_symgs_mv (const sparse_operation_t operation, const
sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 alpha, const
MKL_Complex16 *b, MKL_Complex16 *x, MKL_Complex16 *y);
```

#### Include Files

- mkl_spblas.h

#### Description

The `mkl_sparse_?_symgs_mv` routine performs this operation:

\[
\begin{align*}
x_0 & := x^*\alpha; \\
(L + D) x_1 &= b - U x_0; \\
(U + D) x &= b - L x_1; \\
y & := A x
\end{align*}
\]

where \( A = L + D + U \)
NOTE
This routine is not supported for sparse matrices in BSR, COO, or CSC formats. It supports only the CSR format. Additionally, only symmetric matrices are supported, so the desc.type must be SPARSE_MATRIX_TYPE_SYMMETRIC.

Input Parameters

operation

Specifies the operation performed on input matrix.

SPARSE_OPERATION_NON_TRANSPOSE, op(A) = A.

NOTE
Transpose (SPARSE_OPERATION_TRANSPOSE) and conjugate transpose (SPARSE_OPERATION_CONJUGATE_TRANSPOSE) are not supported.

A

Handle which contains the sparse matrix A.

alpha

Specifies the scalar alpha.

descr

Structure specifying sparse matrix properties.

sparse_matrix_type_t type - Specifies the type of a sparse matrix:

SPARSE_MATRIX_TYPE_GENERAL The matrix is processed as is.

SPARSE_MATRIX_TYPE_SYMMETRIC The matrix is symmetric (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_HERMITIAN The matrix is Hermitian (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_TRIANGULAR The matrix is triangular (only the requested triangle is processed).

SPARSE_MATRIX_TYPE_DIAGONAL The matrix is diagonal (only diagonal elements are processed).

SPARSE_MATRIX_TYPE_BLOCK_TRIANGULAR The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.

SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:

SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.

SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

sparse_diag_type_t diag - Specifies diagonal type for non-general matrices:
SPARSE_DIAG_NON_UNIT  Diagonal elements might not be equal to one.
SPARSE_DIAG_UNIT    Diagonal elements are equal to one.

x
Array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \).
On entry, the array \( x \) must contain the vector \( x \).

b
Array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \).
On entry, the array \( b \) must contain the vector \( b \).

**Output Parameters**

\( x \)
Overwritten by the computed vector \( x \).

\( y \)
Array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \).
Overwritten by the computed vector \( y \).

**Return Values**
The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS  The operation was successful.
SPARSE_STATUS_NOT_INITIALIZED  The routine encountered an empty handle or matrix array.
SPARSE_STATUS_ALLOC_FAILED  Internal memory allocation failed.
SPARSE_STATUS_INVALID_VALUE  The input parameters contain an invalid value.
SPARSE_STATUS_EXECUTION_FAILED  Execution failed.
SPARSE_STATUS_INTERNAL_ERROR  An error in algorithm implementation occurred.
SPARSE_STATUS_NOT_SUPPORTED  The requested operation is not supported.

**mkl_sparse_syrk**

*Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result in a newly allocated sparse matrix.*

**Syntax**

```c
sparse_status_t mkl_sparse_syrk (const sparse_operation_t operation, const sparse_matrix_t A, sparse_matrix_t *C);
```

**Include Files**
- mkl_spblas.h

**Description**
The mkl_sparse_syrk routine performs a sparse matrix-matrix operation which results in a sparse matrix \( C \) that is either Symmetric (real) or Hermitian (complex):

\[ C := A \ast op(A) \]

where \( op(\ast) \) is the transpose for real matrices and conjugate transpose for complex matrices OR

\[ C := op(A) \ast A \]
depending on the matrix modifier \( op \) which can be the transpose for real matrices or conjugate transpose for complex matrices.

Here, \( A \) and \( C \) are sparse matrices.

**NOTE** This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the `mkl_sparse_order` routine before either `mkl_sparse_syrk` or `mkl_sparse_\?_syrkd`.

### Input Parameters

**operation**

Specifies the operation \( op() \) on input matrix.

- `SPARSE_OPERATION_NON_TRANSPOSE`, Non-transpose, \( C := A \cdot op(A) \) where \( op(*) \) is the transpose for real matrices and conjugate transpose for complex matrices
- `SPARSE_OPERATION_TRANSPOSE`, Transpose, \( C := (A^T) \cdot A \) for real matrix \( A \)
- `SPARSE_OPERATION_CONJUGATE_TRANSPOSE`, Conjugate transpose, \( C := (A^H) \cdot A \) for complex matrix \( A \).

**A**

Handle which contains the sparse matrix \( A \).

### Output Parameters

**C**

Handle which contains the resulting sparse matrix. Only the upper-triangular part of the matrix is computed.

### Return Values

The function returns a value indicating whether the operation was successful or not, and why.

- `SPARSE_STATUS_SUCCESS` The operation was successful.
- `SPARSE_STATUS_NOT_INITIALIZED` The routine encountered an empty handle or matrix array.
- `SPARSE_STATUS_ALLOC_FAILED` Internal memory allocation failed.
- `SPARSE_STATUS_INVALID_VALUE` The input parameters contain an invalid value.
- `SPARSE_STATUS_EXECUTION_FAILED` Execution failed.
- `SPARSE_STATUS_INTERNAL_ERROR` An error in algorithm implementation occurred.
- `SPARSE_STATUS_NOT_SUPPORTED` The requested operation is not supported.

### mkl_sparse_\?_syrkd

Computes the product of sparse matrix with its transpose (or conjugate transpose) and stores the result as a dense matrix.

**Syntax**

```c
sparse_status_t mkl_sparse_s_syrkd(sparse_operation_t operation, const sparse_matrix_t A, float alpha, float beta, float *C, sparse_layout_t layout, MKL_INT ldc);
```
sparse_status_t mkl_sparse_d_syrkd (sparse_operation_t operation, const sparse_matrix_t A, double alpha, double beta, double *C, sparse_layout_t layout, MKL_INT ldc);
sparse_status_t mkl_sparse_c_syrkd (sparse_operation_t operation, const sparse_matrix_t A, const MKL_Complex8 alpha, MKL_Complex8 beta, MKL_Complex8 *C, sparse_layout_t layout, MKL_INT ldc);
sparse_status_t mkl_sparse_z_syrkd (sparse_operation_t operation, const sparse_matrix_t A, MKL_Complex16 alpha, MKL_Complex16 beta, MKL_Complex16 *C, sparse_layout_t layout, const MKL_INT ldc);

Include Files

- mkl_spblas.h

Description

The mkl_sparse_?_syrkd routine performs a sparse matrix-matrix operation which results in a dense matrix \( C \) that is either symmetric (real case) or Hermitian (complex case):

\[
C := \beta C + \alpha A \cdot \text{op}(A)
\]

or

\[
C := \beta C + \alpha \cdot \text{op}(A) \cdot A
\]

depending on the matrix modifier \( \text{op} \) which can be the transpose for real matrices or conjugate transpose for complex matrices. Here, \( A \) is a sparse matrix and \( C \) is a dense matrix.

**NOTE** This routine is not supported for sparse matrices in COO or CSC formats. It supports only CSR and BSR formats. Additionally, this routine supports only the sorted CSR and sorted BSR formats for the input matrix. If data is unsorted, call the mkl_sparse_order routine before either mkl_sparse_syrk or mkl_sparse_?_syrkd.

Input Parameters

- **operation**: Specifies the operation \( \text{op()} \) performed on the input matrix.
  
  \( \text{SPARSE_OPERATION_NON_TRANSPOSE} \), Non-transpose, \( C := \beta C + \alpha A \cdot \text{op}(A) \) where \( \text{op}(*) \) is the transpose (real matrices) or conjugate transpose (complex matrices).

  \( \text{SPARSE_OPERATION_TRANSPOSE} \), Transpose, \( C := \beta C + \alpha A^T \cdot A \) for real matrix \( A \).

  \( \text{SPARSE_OPERATION_CONJUGATE_TRANSPOSE} \), Conjugate transpose, \( C := \beta C + \alpha A^H \cdot A \) for complex matrix \( A \).

- **A**: Handle which contains the sparse matrix \( A \).

- **alpha**: Scalar parameter \( \alpha \).

- **beta**: Scalar parameter \( \beta \).

- **layout**: Describes the storage scheme for the dense matrix.
  
  \( \text{layout} = \text{SPARSE_LAYOUT_COLUMN_MAJOR} \) Storage of elements uses column-major layout.
Storage of elements uses row-major layout.

leadc

Leading dimension of matrix C.

NOTE
Only the upper triangular part of matrix C is processed. Therefore, you must set real values of alpha and beta for complex matrices in order to obtain a Hermitian matrix.

Output Parameters

C

Resulting dense matrix. Only the upper triangular part of the matrix is computed.

Return Values

The function returns a value indicating whether the operation was successful or not, and why.

SPARSE_STATUS_SUCCESS
The operation was successful.

SPARSE_STATUS_NOT_INITIALIZED
The routine encountered an empty handle or matrix array.

SPARSE_STATUS_ALLOC_FAILED
Internal memory allocation failed.

SPARSE_STATUS_INVALID_VALUE
The input parameters contain an invalid value.

SPARSE_STATUS_EXECUTION_FAILED
Execution failed.

SPARSE_STATUS_INTERNAL_ERROR
An error in algorithm implementation occurred.

SPARSE_STATUS_NOT_SUPPORTED
The requested operation is not supported.

mkl_sparse_?_dotmv
Computes a sparse matrix-vector product followed by a dot product.

Syntax

sparse_status_t mkl_sparse_s_dotmv (const sparse_operation_t operation, const float alpha, const sparse_matrix_t A, const struct matrix_descr descr, const float *x, const float beta, float *y, float *d);
sparse_status_t mkl_sparse_d_dotmv (const sparse_operation_t operation, const double alpha, const sparse_matrix_t A, const struct matrix_descr descr, const double *x, const double beta, double *y, double *d);
sparse_status_t mkl_sparse_c_dotmv (const sparse_operation_t operation, const MKL_Complex8 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex8 *x, const MKL_Complex8 beta, MKL_Complex8 *y, MKL_Complex8 *d);
sparse_status_t mkl_sparse_z_dotmv (const sparse_operation_t operation, const MKL_Complex16 alpha, const sparse_matrix_t A, const struct matrix_descr descr, const MKL_Complex16 *x, const MKL_Complex16 beta, MKL_Complex16 *y, MKL_Complex16 *d);

Include Files

- mkl_spblas.h
Description
The mkl_sparse_?_dotmv routine computes a sparse matrix-vector product and dot product:

\[ y := \alpha \cdot \text{op}(A) \cdot x + \beta \cdot y \]
\[ d := \sum_i \text{conj}(x_i) \cdot y_i \] (real case)
\[ d := \sum_i x_i \cdot y_i \] (complex case)

where
- \( \alpha \) and \( \beta \) are scalars.
- \( x \) and \( y \) are vectors.
- \( A \) is an \( m \)-by-\( k \) matrix.
- \( \text{conj} \) represents complex conjugation.
- \( \text{op}(A) \) is a matrix modifier.

Available options for \( \text{op}(A) \) are \( A \), \( A^T \), or \( A^H \).

NOTE
For sparse matrices in the BSR format, the supported combinations of \( \text{(indexing,block\_layout)} \) are:
- \( \text{(SPARSE\_INDEX\_BASE\_ZERO, SPARSE\_LAYOUT\_ROW\_MAJOR) } \)
- \( \text{(SPARSE\_INDEX\_BASE\_ONE, SPARSE\_LAYOUT\_COLUMN\_MAJOR) } \)

Input Parameters

- **operation**
  Specifies the operation performed on matrix \( A \).
  - If \( \text{operation} = \text{SPARSE\_OPERATION\_NON\_TRANSPOSE} \), \( \text{op}(A) = A \).
  - If \( \text{operation} = \text{SPARSE\_OPERATION\_TRANSPOSE} \), \( \text{op}(A) = A^T \).
  - If \( \text{operation} = \text{SPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \), \( \text{op}(A) = A^H \).

- **alpha**
  Specifies the scalar \( \alpha \).

- **A**
  Handle which contains the sparse matrix \( A \).

- **descr**
  Structure specifying sparse matrix properties.
  - **type** - Specifies the type of a sparse matrix:
    - \( \text{SPARSE\_MATRIX\_TYPE\_GENERAL} \) - The matrix is processed as is.
    - \( \text{SPARSE\_MATRIX\_TYPE\_SYMMETRIC} \) - The matrix is symmetric (only the requested triangle is processed).
    - \( \text{SPARSE\_MATRIX\_TYPE\_HERMITIAN} \) - The matrix is Hermitian (only the requested triangle is processed).
    - \( \text{SPARSE\_MATRIX\_TYPE\_TRIANGULAR} \) - The matrix is triangular (only the requested triangle is processed).
    - \( \text{SPARSE\_MATRIX\_TYPE\_DIAGONAL} \) - The matrix is diagonal (only diagonal elements are processed).
    - \( \text{SPARSE\_MATRIX\_TYPE\_BLOCK\_TRIANGULAR} \) - The matrix is block-triangular (only requested triangle is processed). Applies to BSR format only.
SPARSE_MATRIX_TYPE_BLOCK_DIAGONAL The matrix is block-diagonal (only diagonal blocks are processed). Applies to BSR format only.

sparse_fill_mode_t mode - Specifies the triangular matrix part for symmetric, Hermitian, triangular, and block-triangular matrices:
- SPARSE_FILL_MODE_LOWER The lower triangular matrix part is processed.
- SPARSE_FILL_MODE_UPPER The upper triangular matrix part is processed.

sparse_diag_type_t diag - Specifies diagonal type for non-general matrices:
- SPARSE_DIAG_NON_UNIT Diagonal elements might not be equal to one.
- SPARSE_DIAG_UNIT Diagonal elements are equal to one.

If operation = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least \( k \), where \( k \) is the number of columns of matrix \( A \).
Otherwise, array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \).
On entry, the array \( x \) must contain the vector \( x \).

beta Specifies the scalar \( \beta \).

If operation = SPARSE_OPERATION_NON_TRANSPOSE, array of size at least \( m \), where \( m \) is the number of rows of matrix \( A \).
Otherwise, array of size at least \( k \), where \( k \) is the number of columns of matrix \( A \).
On entry, the array \( y \) must contain the vector \( y \).

Output Parameters
- \( y \) Overwritten by the updated vector \( y \).
- \( d \) Overwritten by the dot product of \( x \) and \( y \).

Return Values
The function returns a value indicating whether the operation was successful or not, and why.
- SPARSE_STATUS_SUCCESS The operation was successful.
- SPARSE_STATUS_NOT_INITIALIZED The routine encountered an empty handle or matrix array.
- SPARSE_STATUS_ALLOC_FAILED Internal memory allocation failed.
- SPARSE_STATUS_INVALID_VALUE The input parameters contain an invalid value.
- SPARSE_STATUS_EXECUTION_FAILED Execution failed.
- SPARSE_STATUS_INTERNAL_ERROR An error in algorithm implementation occurred.
- SPARSE_STATUS_NOT_SUPPORTED The requested operation is not supported.
## BLAS-like Extensions

provides C and Fortran routines to extend the functionality of the BLAS routines. These include routines to compute vector products, matrix-vector products, and matrix-matrix products.

also provides routines to perform certain data manipulation, including matrix in-place and out-of-place transposition operations combined with simple matrix arithmetic operations. Transposition operations are Copy As Is, Conjugate transpose, Transpose, and Conjugate. Each routine adds the possibility of scaling during the transposition operation by giving some \( \alpha \) and/or \( \beta \) parameters. Each routine supports both row-major orderings and column-major orderings.

Table "BLAS-like Extensions" lists these routines.

The \(<\? >\) symbol in the routine short names is a precision prefix that indicates the data type:

\[
\begin{align*}
  s & \quad \text{float} \\
  d & \quad \text{double} \\
  c & \quad \text{MKL Complex8} \\
  z & \quad \text{MKL Complex16}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Routine</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cblas_?axpby</td>
<td>s, d, c, z</td>
<td>Scales two vectors, adds them to one another and stores result in the vector (routines).</td>
</tr>
<tr>
<td>cblas_?gemmt</td>
<td>s, d, c, z</td>
<td>Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.</td>
</tr>
<tr>
<td>cblas_?gemm3m</td>
<td>c, z</td>
<td>Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.</td>
</tr>
<tr>
<td>cblas_?gemm_batch</td>
<td>s, d, c, z</td>
<td>Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.</td>
</tr>
<tr>
<td>cblas_?gemm3m_batch</td>
<td>c, z</td>
<td>Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.</td>
</tr>
<tr>
<td>mkl_?imatcopy</td>
<td>s, d, c, z</td>
<td>Performs scaling and in-place transposition/copying of matrices.</td>
</tr>
<tr>
<td>mkl_?omatcopy</td>
<td>s, d, c, z</td>
<td>Performs scaling and out-of-place transposition/copying of matrices.</td>
</tr>
<tr>
<td>mkl_?omatcopy2</td>
<td>s, d, c, z</td>
<td>Performs two-strided scaling and out-of-place transposition/copying of matrices.</td>
</tr>
<tr>
<td>mkl_?omatadd</td>
<td>s, d, c, z</td>
<td>Performs scaling and sum of two matrices including their out-of-place transposition/copying.</td>
</tr>
<tr>
<td>cblas_?gemm_pack_get_size</td>
<td>s, d</td>
<td>Returns the number of bytes required to store the packed matrix.</td>
</tr>
<tr>
<td>cblas_?gemm_*_pack_get_size</td>
<td>integer</td>
<td>Returns the number of bytes required to store the packed matrix.</td>
</tr>
<tr>
<td>cblas_?gemm_alloc</td>
<td>s, d</td>
<td>Allocates storage for a packed matrix.</td>
</tr>
</tbody>
</table>
### Routine | Data Types | Description
--- | --- | ---
`cblas_?gemm_pack` | s, d | Performs scaling and packing of the matrix into the previously allocated buffer.
`cblas_gemm_*_pack` | integer | Pack the matrix into the buffer allocated previously.
`cblas_?gemm_compute` | s, d | Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
`cblas_gemm_*_compute` | integer | Computes a matrix-matrix product with general integer matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
`cblas_?gemm_free` | s, d | Frees the storage previously allocated for the packed matrix.
`cblas_gemm_*` | integer | Computes a matrix-matrix product with general integer matrices.

### cblas_?axpby

**Scales two vectors, adds them to one another and stores result in the vector.**

#### Syntax

```c
void cblas_saxpby (const MKL_INT n, const float a, const float *x, const MKL_INT incx, const float b, float *y, const MKL_INT incy);
void cblas_daxpby (const MKL_INT n, const double a, const double *x, const MKL_INT incx, const double b, double *y, const MKL_INT incy);
void cblas_caxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx, const void *b, void *y, const MKL_INT incy);
void cblas_zaxpby (const MKL_INT n, const void *a, const void *x, const MKL_INT incx, const void *b, void *y, const MKL_INT incy);
```

#### Include Files

- mkl.h

#### Description

The ?axpby routines perform a vector-vector operation defined as

\[ y := a \cdot x + b \cdot y \]

where:

- `a` and `b` are scalars
- `x` and `y` are vectors each with `n` elements.

#### Input Parameters

- `n` Specifies the number of elements in vectors `x` and `y`. 
a  
Specifies the scalar \( a \).

x  
Array, size at least \((1 + (n-1) \times \text{abs(incx)})\).

\( \text{incx} \)  
Specifies the increment for the elements of \( x \).

b  
Specifies the scalar \( b \).

y  
Array, size at least \((1 + (n-1) \times \text{abs(incy)})\).

\( \text{incy} \)  
Specifies the increment for the elements of \( y \).

Output Parameters

\( y \)  
Contains the updated vector \( y \).

Example

For examples of routine usage, see these code examples in the Intel® MKL installation directory:

- cblas_saxpby: examples\cblas\source\cblas_saxpbyx.c
- cblas_daxpby: examples\cblas\source\cblas_daxpbyx.c
- cblas_caxpby: examples\cblas\source\cblas_caxpbyx.c
- cblas_zaxpby: examples\cblas\source\cblas_zaxpbyx.c

\textbf{cblas} \_\textbf{?}gemmt}  
\textit{Computes a matrix-matrix product with general matrices but updates only the upper or lower triangular part of the result matrix.}

\textbf{Syntax}

\begin{verbatim}
void cblas_sgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const 
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k, 
const float alpha, const float *a, const MKL_INT lda, const float *b, const MKL_INT 
ldb, const float beta, float *c, const MKL_INT ldc);

void cblas_dgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const 
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k, 
const double alpha, const double *a, const MKL_INT lda, const double *b, const MKL_INT 
ldb, const double beta, double *c, const MKL_INT ldc);

void cblas_cgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const 
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k, 
const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT 
ldb, const void *beta, void *c, const MKL_INT ldc);

void cblas_zgemmt (const CBLAS_LAYOUT Layout, const CBLAS_UPLO uplo, const 
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT n, const MKL_INT k, 
const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT 
ldb, const void *beta, void *c, const MKL_INT ldc);
\end{verbatim}

\textbf{Include Files}

- mkl.h
Description

The ?gemmt routines compute a scalar-matrix-matrix product with general matrices and add the result to the upper or lower part of a scalar-matrix product. These routines are similar to the ?gemm routines, but they only access and update a triangular part of the square result matrix (see Application Notes below).

The operation is defined as

\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C, \]

where:

- \( \text{op}(X) \) is one of \( \text{op}(X) = X \), or \( \text{op}(X) = X^T \), or \( \text{op}(X) = X^H \),
- \( \alpha \) and \( \beta \) are scalars,
- \( A \), \( B \) and \( C \) are matrices:
  - \( \text{op}(A) \) is an \( n \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( n \)-by-\( n \) upper or lower triangular matrix.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **uplo**
  Specifies whether the upper or lower triangular part of the array \( c \) is used. If \( \text{uplo} = \text{'CblasUpper'} \), then the upper triangular part of the array \( c \) is used. If \( \text{uplo} = \text{'CblasLower'} \), then the lower triangular part of the array \( c \) is used.

- **transa**
  Specifies the form of \( \text{op}(A) \) used in the matrix multiplication:
  - if \( \text{transa} = \text{'CblasNoTrans'} \), then \( \text{op}(A) = A \);
  - if \( \text{transa} = \text{'CblasTrans'} \), then \( \text{op}(A) = A^T \);
  - if \( \text{transa} = \text{'CblasConjTrans'} \), then \( \text{op}(A) = A^H \).

- **transb**
  Specifies the form of \( \text{op}(B) \) used in the matrix multiplication:
  - if \( \text{transb} = \text{'CblasNoTrans'} \), then \( \text{op}(B) = B \);
  - if \( \text{transb} = \text{'CblasTrans'} \), then \( \text{op}(B) = B^T \);
  - if \( \text{transb} = \text{'CblasConjTrans'} \), then \( \text{op}(B) = B^H \).

- **n**
  Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

- **k**
  Specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

- **alpha**
  Specifies the scalar \( \alpha \).

- **a**
<table>
<thead>
<tr>
<th>transa='CblasNoTrans'</th>
<th>transa='CblasTrans'</th>
<th>transa='CblasConjTrans'</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>or</td>
<td>or</td>
</tr>
<tr>
<td>'CblasNoTrans'</td>
<td>'CblasTrans'</td>
<td>'CblasConjTrans'</td>
</tr>
</tbody>
</table>
\[
\begin{array}{|l|l|l|}
\hline
\text{Layout}= \text{'CblasColMaj or' } & \text{Array, size } lda * k. & \text{Array, size } lda * n. \\
& \text{Before entry, the leading } n\text{-by-}k \text{ part of the array } a \text{ must contain the matrix } A. & \text{Before entry, the leading } k\text{-by-}n \text{ part of the array } a \text{ must contain the matrix } A. \\
\hline
\text{Layout}= \text{'CblasRowMaj or' } & \text{Array, size } lda * n. & \text{Array, size } lda * k. \\
& \text{Before entry, the leading } k\text{-by-}n \text{ part of the array } a \text{ must contain the matrix } A. & \text{Before entry, the leading } n\text{-by-}k \text{ part of the array } a \text{ must contain the matrix } A. \\
\hline
\end{array}
\]

\text{lda}

Specifies the leading dimension of } a \text{ as declared in the calling (sub)program.

\[
\begin{array}{|l|l|}
\hline
\text{transa}= \text{'CblasNoTrans' } & \text{transa}= \text{'CblasTrans' or 'CblasConjTrans' } \\
\hline
\text{Layout}= \text{'CblasColMaj or' } & \text{lda must be at least } \text{max}(1, n). & \text{lda must be at least } \text{max}(1, k). \\
\hline
\text{Layout}= \text{'CblasRowMaj or' } & \text{lda must be at least } \text{max}(1, k). & \text{lda must be at least } \text{max}(1, n). \\
\hline
\end{array}
\]

\text{b}

\[
\begin{array}{|l|l|}
\hline
\text{transb}= \text{'CblasNoTrans' } & \text{transb}= \text{'CblasTrans' or 'CblasConjTrans' } \\
\hline
\text{Layout}= \text{'CblasColMaj or' } & \text{Array, size } ldb * n. & \text{Array, size } ldb * k. \\
& \text{Before entry, the leading } n\text{-by-}k \text{ part of the array } b \text{ must contain the matrix } B. & \text{Before entry, the leading } k\text{-by-}n \text{ part of the array } b \text{ must contain the matrix } B. \\
\hline
\text{Layout}= \text{'CblasRowMaj or' } & \text{Array, size } ldb * k. & \text{Array, size } ldb * n. \\
& \text{Before entry, the leading } k\text{-by-}n \text{ part of the array } b \text{ must contain the matrix } B. & \text{Before entry, the leading } n\text{-by-}k \text{ part of the array } b \text{ must contain the matrix } B. \\
\hline
\end{array}
\]

\text{ldb}

Specifies the leading dimension of } b \text{ as declared in the calling (sub)program.

\[
\begin{array}{|l|l|}
\hline
\text{transb}= \text{'CblasNoTrans' } & \text{transb}= \text{'CblasTrans' or 'CblasConjTrans' } \\
\hline
\text{Layout}= \text{'CblasColMaj or' } & \text{ldb must be at least } \text{max}(1, k). & \text{ldb must be at least } \text{max}(1, n). \\
\hline
\text{Layout}= \text{'CblasRowMaj or' } & \text{ldb must be at least } \text{max}(1, n). & \text{ldb must be at least } \text{max}(1, k). \\
\hline
\end{array}
\]

\text{beta}

Specifies the scalar } beta \text{. When } beta \text{ is equal to zero, then } c \text{ need not be set on input.

\text{c}

Array, size } ldc \text{ by } n.
When \( \beta \) is equal to zero, \( c \) need not be set on input.

<table>
<thead>
<tr>
<th>( \text{uplo} = 'CblasUpper' )</th>
<th>( \text{uplo} = 'CblasLower' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>The leading ( n )-by-( n ) upper triangular part of the array ( c ) must contain the upper triangular part of the matrix ( C ) and the strictly lower triangular part of ( c ) is not referenced.</td>
<td>The leading ( n )-by-( n ) lower triangular part of the array ( c ) must contain the lower triangular part of the matrix ( C ) and the strictly upper triangular part of ( c ) is not referenced.</td>
</tr>
</tbody>
</table>

\( \text{ldc} \)

Specifies the leading dimension of \( c \) as declared in the calling (sub)program. The value of \( \text{ldc} \) must be at least \( \max(1, n) \).

### Output Parameters

\( c \)

- When \( \text{uplo} = 'CblasUpper' \), the upper triangular part of the array \( c \) is overwritten by the upper triangular part of the updated matrix.
- When \( \text{uplo} = 'CblasLower' \), the lower triangular part of the array \( c \) is overwritten by the lower triangular part of the updated matrix.

### Application Notes

These routines only access and update the upper or lower triangular part of the result matrix. This can be useful when the result is known to be symmetric; for example, when computing a product of the form \( C := \alpha A S B^T + \beta C \), where \( S \) and \( C \) are symmetric matrices and \( B \) is a general matrix. In this case, first compute \( A := B S \) (which can be done using the corresponding \(?\text{symm}\) routine), then compute \( C := \alpha A B^T + \beta C \) using the \(?\text{gemm}\) routine.

### \text{cblas}\_?gemm3m

**Computes a scalar-matrix-matrix product using matrix multiplications and adds the result to a scalar-matrix product.**

### Syntax

```c
void cblas_cgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);

void cblas_zgemm3m (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *alpha, const void *a, const MKL_INT lda, const void *b, const MKL_INT ldb, const void *beta, void *c, const MKL_INT ldc);
```

### Include Files

- \text{mkl.h}

### Description

The \(?\text{gemm3m}\) routines perform a matrix-matrix operation with general complex matrices. These routines are similar to the \(?\text{gemm}\) routines, but they use fewer matrix multiplication operations (see \text{Application Notes} below).
The operation is defined as
\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C, \]

where:
- \( \text{op}(x) \) is one of \( \text{op}(x) = x \), or \( \text{op}(x) = x' \), or \( \text{op}(x) = \text{conjg}(x') \),
- \( \alpha \) and \( \beta \) are scalars,
- \( A \), \( B \), and \( C \) are matrices:
  - \( \text{op}(A) \) is an \( m \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.

**Input Parameters**

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**transa**

Specifies the form of \( \text{op}(A) \) used in the matrix multiplication:
- if \( \text{transa} = \text{CblasNoTrans} \), then \( \text{op}(A) = A \);
- if \( \text{transa} = \text{CblasTrans} \), then \( \text{op}(A) = A' \);
- if \( \text{transa} = \text{CblasConjTrans} \), then \( \text{op}(A) = \text{conjg}(A') \).

**transb**

Specifies the form of \( \text{op}(B) \) used in the matrix multiplication:
- if \( \text{transb} = \text{CblasNoTrans} \), then \( \text{op}(B) = B \);
- if \( \text{transb} = \text{CblasTrans} \), then \( \text{op}(B) = B' \);
- if \( \text{transb} = \text{CblasConjTrans} \), then \( \text{op}(B) = \text{conjg}(B') \).

**m**

Specifies the number of rows of the matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

**n**

Specifies the number of columns of the matrix \( \text{op}(B) \) and the number of columns of the matrix \( C \).

The value of \( n \) must be at least zero.

**k**

 Specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \).

The value of \( k \) must be at least zero.

**alpha**

Specifies the scalar \( \alpha \).

**a**

<table>
<thead>
<tr>
<th>transa=CblasNoTrans</th>
<th>transa=CblasTrans or transa=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout = CblasColMajor</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Array, size lda*k.</strong></td>
<td></td>
</tr>
<tr>
<td>Before entry, the leading ( m )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
<td></td>
</tr>
<tr>
<td><strong>Array, size lda*m.</strong></td>
<td></td>
</tr>
<tr>
<td>Before entry, the leading ( k )-by-( m ) part of the array ( a ) must contain the matrix ( A ).</td>
<td></td>
</tr>
</tbody>
</table>
Before entry, the leading $k$-by-$m$ part of the array $a$ must contain the matrix $A$. Before entry, the leading $m$-by-$k$ part of the array $a$ must contain the matrix $A$.

Specifies the leading dimension of $a$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
<th>Array, size $lda \times m$. Before entry, the leading $k$-by-$m$ part of the array $a$ must contain the matrix $A$.</th>
<th>Array, size $lda \times k$. Before entry, the leading $m$-by-$k$ part of the array $a$ must contain the matrix $A$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>transa = CblasNoTrans</td>
<td>transa = CblasTrans or transa = CblasConjTrans</td>
<td></td>
</tr>
<tr>
<td>Layout = CblasColMajor</td>
<td>$lda$ must be at least $\max(1, m)$.</td>
<td>$lda$ must be at least $\max(1, k)$.</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td>$lda$ must be at least $\max(1, k)$.</td>
<td>$lda$ must be at least $\max(1, m)$.</td>
</tr>
</tbody>
</table>

Before entry, the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$. Before entry the leading $n$-by-$k$ part of the array $b$ must contain the matrix $B$.

Specifies the leading dimension of $b$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>Array, size $ldb \times n$. Before entry, the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$.</th>
<th>Array, size $ldb \times k$. Before entry the leading $n$-by-$k$ part of the array $b$ must contain the matrix $B$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>transb = CblasNoTrans</td>
<td>transb = CblasTrans or transb = CblasConjTrans</td>
<td></td>
</tr>
<tr>
<td>Layout = CblasColMajor</td>
<td>$ldb$ must be at least $\max(1, k)$.</td>
<td>$ldb$ must be at least $\max(1, n)$.</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td>$ldb$ must be at least $\max(1, n)$.</td>
<td>$ldb$ must be at least $\max(1, k)$.</td>
</tr>
</tbody>
</table>

Specifies the scalar $beta$.

When $beta$ is equal to zero, then $c$ need not be set on input.
c

<table>
<thead>
<tr>
<th>Layout</th>
<th>Array, size ldc by n. Before entry, the leading m-by-n part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout</th>
<th>Array, size ldc by m. Before entry, the leading n-by-m part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasRowMajor</td>
<td></td>
</tr>
</tbody>
</table>

ldc

Specifies the leading dimension of c as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout</th>
<th>ldc must be at least max(1, m).</th>
<th>ldc must be at least max(1, n).</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

- c
  Overwritten by the m-by-n matrix \((\alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C)\).

Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

\[ f_l(x \text{ op } y) = (x \text{ op } y)(1 + \delta), |\delta| \leq u, \text{ op} = \times, /, f_l(x \pm y) = x(1 + \alpha) \pm y(1 + \beta), |\alpha|, |\beta| \leq u \]

then for an n-by-n matrix \( \hat{C} = f_l(C_1 + iC_2) = f_l((A_1 + iA_2)(B_1 + iB_2)) = C_1 + iC_2 \), the following bounds are satisfied:

\[ \|C_1 - \hat{C}_1\| \leq 2(n+1)u\|A\|_\infty\|B\|_\infty + O(u^2), \]
\[ \|C_2 - \hat{C}_2\| \leq 4(n+4)u\|A\|_\infty\|B\|_\infty + O(u^2), \]

where \( \|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty) \), and \( \|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty) \).

Thus the corresponding matrix multiplications are stable.

cblas_?gemm_batch

*Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.*

Syntax

```c
void cblas_sgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array, const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array, const MKL_INT* k_array, const float* alpha_array, const float **a_array, const MKL_INT*...
```
lda_array, const float **b_array, const MKL_INT* ldb_array, const float* beta_array,
float **c_array, const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
void cblas_dgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const double* alpha_array, const double **a_array, const MKL_INT* ldda_array,
const double **b_array, const MKL_INT* ldb_array, const double* beta_array, double **c_array,
const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
void cblas_cgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const void* alpha_array, const void **a_array, const MKL_INT* lda_array,
const void **b_array, const MKL_INT* ldb_array, const void *beta_array, void **c_array,
const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);
void cblas_zgemm_batch (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE* transa_array,
const CBLAS_TRANSPOSE* transb_array, const MKL_INT* m_array, const MKL_INT* n_array,
const MKL_INT* k_array, const void* alpha_array, const void **a_array, const MKL_INT* lda_array,
const void **b_array, const MKL_INT* ldb_array, const void *beta_array, void **c_array,
const MKL_INT* ldc_array, const MKL_INT group_count, const MKL_INT*
group_size);

Include Files
• mkl.h

Description
The ?gemm_batch routines perform a series of matrix-matrix operations with general matrices. They are
similar to the ?gemm routine counterparts, but the ?gemm_batch routines perform matrix-matrix operations
with groups of matrices, processing a number of groups at once. The groups contain matrices with the same
parameters.

The operation is defined as

```c
idx = 0
for i = 0..group_count - 1
    alpha and beta in alpha_array[i] and beta_array[i]
    for j = 0..group_size[i] - 1
        A, B, and C matrix in a_array[idx], b_array[idx], and c_array[idx]
        C := alpha*op(A)*op(B) + beta*C,
        idx = idx + 1
    end for
end for
```

where:

- \( \text{op}(X) \) is one of \( \text{op}(X) = X, \text{op}(X) = X^T, \) or \( \text{op}(X) = X^H, \)
- \( \alpha \) and \( \beta \) are scalar elements of \( \text{alpha_array} \) and \( \text{beta_array} \),
- \( A, B \) and \( C \) are matrices such that for \( m, n, \) and \( k \) which are elements of \( m_array, n_array, \) and \( k_array \):
  - \( \text{op}(A) \) is an \( m \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.
A, B, and C represent matrices stored at addresses pointed to by a_array, b_array, and c_array, respectively. The number of entries in a_array, b_array, and c_array is total_batch_count = the sum of all of the group_size entries.

See also gemm for a detailed description of multiplication for general matrices and ?gemm3m_batch, BLAS-like extension routines for similar matrix-matrix operations.

NOTE
Error checking is not performed for Windows* single dynamic libraries for the ?gemm_batch routines.

Input Parameters

Layout
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

tansa_array
Array of size group_count. For the group i, transa_i = transa_array[i] specifies the form of op(A) used in the matrix multiplication:

if transa_i = CblasNoTrans, then op(A) = A;
if transa_i = CblasTrans, then op(A) = A^T;
if transa_i = CblasConjTrans, then op(A) = A^H.

transb_array
Array of size group_count. For the group i, transb_i = transb_array[i] specifies the form of op(B_i) used in the matrix multiplication:

if transb_i = CblasNoTrans, then op(B) = B;
if transb_i = CblasTrans, then op(B) = B^T;
if transb_i = CblasConjTrans, then op(B) = B^H.

m_array
Array of size group_count. For the group i, m_i = m_array[i] specifies the number of rows of the matrix op(A) and of the matrix C.

The value of each element of m_array must be at least zero.

n_array
Array of size group_count. For the group i, n_i = n_array[i] specifies the number of columns of the matrix op(B) and the number of columns of the matrix C.

The value of each element of n_array must be at least zero.

k_array
Array of size group_count. For the group i, k_i = k_array[i] specifies the number of columns of the matrix op(A) and the number of rows of the matrix op(B).

The value of each element of k_array must be at least zero.

alpha_array
Array of size group_count. For the group i, alpha_array[i] specifies the scalar alpha_i.

a_array
Array, size total_batch_count, of pointers to arrays used to store A matrices.
lda_array

Array of size group_count. For the group i, lda_i = lda_array[i] specifies the leading dimension of the array storing matrix A as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th></th>
<th>transa_i=CblasNoTrans</th>
<th>transa_i=CblasTrans or transa_i=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasColMajor</td>
<td>lda_i must be at least max(1, m_i).</td>
<td>lda_i must be at least max(1, k_i).</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td>lda_i must be at least max(1, k_i).</td>
<td>lda_i must be at least max(1, m_i).</td>
</tr>
</tbody>
</table>

b_array

Array, size total_batch_count, of pointers to arrays used to store B matrices.

ldb_array

Array of size group_count. For the group i, ldb_i = ldb_array[i] specifies the leading dimension of the array storing matrix B as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th></th>
<th>transb_i=CblasNoTrans</th>
<th>transb_i=CblasTrans or transb_i=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasColMajor</td>
<td>ldb_i must be at least max(1, k_i).</td>
<td>ldb_i must be at least max(1, n_i).</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td>ldb_i must be at least max(1, n_i).</td>
<td>ldb_i must be at least max(1, k_i).</td>
</tr>
</tbody>
</table>

beta_array

Array of size group_count. For the group i, beta_array[i] specifies the scalar beta_i.

When beta_i is equal to zero, then C matrices in group i need not be set on input.

c_array

Array, size total_batch_count, of pointers to arrays used to store C matrices.

ldc_array

Array of size group_count. For the group i, ldc_i = ldc_array[i] specifies the leading dimension of all arrays storing matrix C in group i as declared in the calling (sub)program.

When Layout = CblasColMajor ldc_i must be at least max(1, m_i).

When Layout = CblasRowMajor ldc_i must be at least max(1, n_i).

group_count

Specifies the number of groups. Must be at least 0.

group_size

Array of size group_count. The element group_size[i] specifies the number of matrices in group i. Each element in group_size must be at least 0.

Output Parameters

c_array

Output buffer, overwritten by total_batch_count matrix multiply operations of the form alpha*op(A)*op(B) + beta*C.
**cblas_?gemm3m_batch**

Computes scalar-matrix-matrix products and adds the results to scalar matrix products for groups of general matrices.

**Syntax**

```c
void cblas_cgemm3m_batch
(const CBLAS_LAYOUT Layout,
 const CBLAS_TRANSPOSE* transa_array,
 const CBLAS_TRANSPOSE* transb_array,
 const MKL_INT* m_array,
 const MKL_INT* n_array,
 const MKL_INT* k_array,
 const void* alpha_array,
 const void** a_array,
 const void** b_array,
 const void** c_array,
 const MKL_INT* ld_a_array,
 const MKL_INT* ld_b_array,
 const MKL_INT* ld_c_array,
 const MKL_INT group_count,
 const MKL_INT* group_size);
```

```c
void cblas_zgemm3m_batch
(const CBLAS_LAYOUT Layout,
 const CBLAS_TRANSPOSE* transa_array,
 const CBLAS_TRANSPOSE* transb_array,
 const MKL_INT* m_array,
 const MKL_INT* n_array,
 const MKL_INT* k_array,
 const void* alpha_array,
 const void** a_array,
 const void** b_array,
 const void** c_array,
 const MKL_INT* ld_a_array,
 const MKL_INT* ld_b_array,
 const MKL_INT* ld_c_array,
 const MKL_INT group_count,
 const MKL_INT* group_size);
```

**Include Files**

- mkl.h

**Description**

The ?gemm3m_batch routines perform a series of matrix-matrix operations with general matrices. They are similar to the ?gemm3m routine counterparts, but the ?gemm3m_batch routines perform matrix-matrix operations with groups of matrices, processing a number of groups at once. The groups contain matrices with the same parameters. The ?gemm3m_batch routines use fewer matrix multiplications than the ?gemm_batch routines, as described in the Application Notes.

The operation is defined as

```c
idx = 0
for i = 0..group_count - 1
    alpha and beta in alpha_array[i] and beta_array[i]
    for j = 0..group_size[i] - 1
        A, B, and C matrix in a_array[idx], b_array[idx], and c_array[idx]
        C := alpha*op(A)*op(B) + beta*C,
        idx = idx + 1
    end for
end for
```

where:

- \( \text{op}(X) \) is one of \( \text{op}(X) = X, \text{op}(X) = X^T, \text{or} \ \text{op}(X) = X^H \),
- \( \alpha \) and \( \beta \) are scalar elements of \( \text{alpha_array} \) and \( \text{beta_array} \),
- \( A, B \) and \( C \) are matrices such that for \( m, n, \) and \( k \) which are elements of \( \text{m_array}, \text{n_array}, \) and \( \text{k_array} \):
  - \( \text{op}(A) \) is an \( m\times k \) matrix,
  - \( \text{op}(B) \) is a \( k\times n \) matrix,
  - \( C \) is an \( m\times n \) matrix.
$A$, $B$, and $C$ represent matrices stored at addresses pointed to by $a\_array$, $b\_array$, and $c\_array$, respectively. The number of entries in $a\_array$, $b\_array$, and $c\_array$ is $total\_batch\_count = \text{the sum of all the group size entries}$.

See also gemm for a detailed description of multiplication for general matrices and gemm\_batch, BLAS-like extension routines for similar matrix-matrix operations.

**NOTE**

Error checking is not performed for Intel® MKL Windows® single dynamic libraries for the ?gemm3m\_batch routines.

### Input Parameters

**Layout**

Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

**transa\_array**

Array of size $group\_count$. For the group $i$, $transa_i = transa\_array[i]$ specifies the form of $\text{op} (A)$ used in the matrix multiplication:

- if $transa_i = \text{CblasNoTrans}$, then $\text{op} (A) = A$;
- if $transa_i = \text{CblasTrans}$, then $\text{op} (A) = A^T$;
- if $transa_i = \text{CblasConjTrans}$, then $\text{op} (A) = A^H$.

**transb\_array**

Array of size $group\_count$. For the group $i$, $transb_i = transb\_array[i]$ specifies the form of $\text{op} (B_i)$ used in the matrix multiplication:

- if $transb_i = \text{CblasNoTrans}$, then $\text{op} (B) = B$;
- if $transb_i = \text{CblasTrans}$, then $\text{op} (B) = B^T$;
- if $transb_i = \text{CblasConjTrans}$, then $\text{op} (B) = B^H$.

**m\_array**

Array of size $group\_count$. For the group $i$, $m_i = m\_array[i]$ specifies the number of rows of the matrix $\text{op} (A)$ and of the matrix $C$.

The value of each element of $m\_array$ must be at least zero.

**n\_array**

Array of size $group\_count$. For the group $i$, $n_i = n\_array[i]$ specifies the number of columns of the matrix $\text{op} (B)$ and the number of columns of the matrix $C$.

The value of each element of $n\_array$ must be at least zero.

**k\_array**

Array of size $group\_count$. For the group $i$, $k_i = k\_array[i]$ specifies the number of columns of the matrix $\text{op} (A)$ and the number of rows of the matrix $\text{op} (B)$.

The value of each element of $k\_array$ must be at least zero.

**alpha\_array**

Array of size $group\_count$. For the group $i$, $alpha\_array[i]$ specifies the scalar $alpha_i$.

**a\_array**

Array, size $total\_batch\_count$, of pointers to arrays used to store $A$ matrices.
lda_array

Array of size `group_count`. For the group `i`, `lda_i = lda_array[i]` specifies the leading dimension of the array storing matrix `A` as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>transa&lt;sub&gt;i&lt;/sub&gt;=CblasNoTrans</th>
<th>transa&lt;sub&gt;i&lt;/sub&gt;=CblasTrans or transa&lt;sub&gt;i&lt;/sub&gt;=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasColMajor</td>
<td><code>lda_i</code> must be at least <code>max(1, m_i)</code>.</td>
</tr>
<tr>
<td></td>
<td><code>lda_i</code> must be at least <code>max(1, k_i)</code>.</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td><code>lda_i</code> must be at least <code>max(1, k_i)</code>.</td>
</tr>
<tr>
<td></td>
<td><code>lda_i</code> must be at least <code>max(1, m_i)</code>.</td>
</tr>
</tbody>
</table>

b_array

Array, size `total_batch_count`, of pointers to arrays used to store `B` matrices.

lbd_array

Array of size `group_count`. For the group `i`, `ldb_i = ldb_array[i]` specifies the leading dimension of the array storing matrix `B` as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>transb&lt;sub&gt;i&lt;/sub&gt;=CblasNoTrans</th>
<th>transb&lt;sub&gt;i&lt;/sub&gt;=CblasTrans or transb&lt;sub&gt;i&lt;/sub&gt;=CblasConjTrans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasColMajor</td>
<td><code>ldb_i</code> must be at least <code>max(1, k_i)</code>.</td>
</tr>
<tr>
<td></td>
<td><code>ldb_i</code> must be at least <code>max(1, n_i)</code>.</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td><code>ldb_i</code> must be at least <code>max(1, n_i)</code>.</td>
</tr>
<tr>
<td></td>
<td><code>ldb_i</code> must be at least <code>max(1, k_i)</code>.</td>
</tr>
</tbody>
</table>

beta_array

For the group `i`, `beta_array[i]` specifies the scalar `beta<sub>i</sub>`. When `beta<sub>i</sub>` is equal to zero, then `C` matrices in group `i` need not be set on input.

c_array

Array, size `total_batch_count`, of pointers to arrays used to store `C` matrices.

ldc_array

Array of size `group_count`. For the group `i`, `ldc<sub>i</sub> = ldc_array[i]` specifies the leading dimension of all arrays storing matrix `C` in group `i` as declared in the calling (sub)program.

When `Layout = CblasColMajor` `ldc<sub>i</sub>` must be at least `max(1, m_i)`. When `Layout = CblasRowMajor` `ldc<sub>i</sub>` must be at least `max(1, n_i)`.

group_count

Specifies the number of groups. Must be at least 0.

group_size

Array of size `group_count`. The element `group_size[i]` specifies the number of matrices in group `i`. Each element in `group_size` must be at least 0.

Output Parameters

`c_array`

Overwritten by the `m_i`-by-`n_i` matrix `(alpha<sub>i</sub>*op(A)*op(B) + beta<sub>i</sub>*C)` for group `i`. 
Application Notes

These routines perform a complex matrix multiplication by forming the real and imaginary parts of the input matrices. This uses three real matrix multiplications and five real matrix additions instead of the conventional four real matrix multiplications and two real matrix additions. The use of three real matrix multiplications reduces the time spent in matrix operations by 25%, resulting in significant savings in compute time for large matrices.

If the errors in the floating point calculations satisfy the following conditions:

\[ fl(x \text{ op} y) = (x \text{ op} y)(1+\delta), |\delta| \leq u, \quad \text{op=x, /, fl}(x \pm y) = x(1+\alpha) \pm y(1+\beta), |\alpha|, |\beta| \leq u \]

then for an \( n \)-by-\( n \) matrix \( \hat{C} = fl(C_1 + iC_2) = fl((A_1 + iA_2)(B_1 + iB_2)) = \hat{C}_1 + i\hat{C}_2 \), the following bounds are satisfied:

\[ \| \hat{C}_1 - C_1 \| \leq 2(n+1)u \|A\|_\infty \|B\|_\infty + O(u^2), \]
\[ \| \hat{C}_2 - C_2 \| \leq 4(n+4)u \|A\|_\infty \|B\|_\infty + O(u^2), \]

where \( \|A\|_\infty = \max(\|A_1\|_\infty, \|A_2\|_\infty) \), and \( \|B\|_\infty = \max(\|B_1\|_\infty, \|B_2\|_\infty) \).

Thus the corresponding matrix multiplications are stable.

cblas_?trsm_batch

Solves a triangular matrix equation for a group of matrices.

Syntax

void cblas_strsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *TransA_Array, const CBLAS_DIAG *Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const float *alpha_Array, const float *A_Array, const MKL_INT *lda_Array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_dtrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const double *alpha_Array, const double *A_Array, const MKL_INT *lda_Array, double *B_Array, const MKL_INT *ldb_Array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_ctrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const void *alpha_Array, const void *A_Array, const MKL_INT *lda_Array, void *B_Array, const MKL_INT *ldb_Array, const MKL_INT group_count, const MKL_INT *group_size);

void cblas_ztrsm_batch (const CBLAS_LAYOUT Layout, const CBLAS_SIDE *Side_Array, const CBLAS_UPLO *Uplo_Array, const CBLAS_TRANSPOSE *Transa_Array, const CBLAS_DIAG *Diag_Array, const MKL_INT *M_Array, const MKL_INT *N_Array, const void *alpha_Array, const void *A_Array, const MKL_INT *lda_Array, void *B_Array, const MKL_INT *ldb_Array, const MKL_INT group_count, const MKL_INT *group_size);

Include Files

- mkl.h

Description

BLAS and Sparse BLAS Routines
The `trsm_batch` routines solve a series of matrix equations. They are similar to the `trsm` routines except that they operate on groups of matrices which have the same parameters. The `trsm_batch` routines process a number of groups at once.

```plaintext
idx = 0
for i = 0..group_count - 1
    alpha in alpha_array[i]
    for j = 0..group_size[i] - 1
        A and B matrix in a_array[idx] and b_array[idx]
        Solve op(A)*X = alpha*B
        or
        Solve X*op(A) = alpha*B
        idx = idx + 1
    end for
end for
```

where:

- `alpha` is a scalar element of `alpha_array`,
- `X` and `B` are `m`-by-`n` matrices for `m` and `n` which are elements of `m_array` and `n_array`, respectively,
- `A` is a unit, or non-unit, upper or lower triangular matrix,
- and `op(A)` is one of `op(A) = A`, or `op(A) = A^T`, or `op(A) = conjg(A^T)`.
- `A` and `B` represent matrices stored at addresses pointed to by `a_array` and `b_array`, respectively. There are `total_batch_count` entries in each of `a_array` and `b_array`, where `total_batch_count` is the sum of all the `group_size` entries.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **side_array**
  Array of size `group_count`. For group `i`, `0 ≤ i < group_count`, `side[i]` specifies whether `op(A)` appears on the left or right of `X` in the equation:
  - if `side[i]` = CblasLeft, then `op(A)*X = alpha*B`;
  - if `side[i]` = CblasRight, then `X*op(A) = alpha*B`.

- **uplo_array**
  Array of size `group_count`. For group `i`, `0 ≤ i < group_count`, `uplo[i]` specifies whether the matrix `A` is upper or lower triangular:
  - `uplo[i]` = CblasUpper if `uplo[i]` = CblasLower, then the matrix is lower triangular.

- **transa_array**
  Array of size `group_count`. For group `i`, `0 ≤ i < group_count`, `transa[i]` specifies the form of `op(A)` used in the matrix multiplication:
  - if `transa[i]` = CblasNoTrans, then `op(A) = A`;
  - if `transa[i]` = CblasTrans;
  - if `transa[i]` = CblasConjTrans, then `op(A) = conjg(A')`.

- **diag_array**
  Array of size `group_count`. For group `i`, `0 ≤ i < group_count`, `diag[i]` specifies whether the matrix `A` is unit triangular:
  - if `diag[i]` = CblasUnit then the matrix is unit triangular;
if $diag_i = \text{CblasNonUnit}$, then the matrix is not unit triangular.

$m\_array$

Array of size $group\_count$. For group $i$, $0 \leq i < group\_count$, $m_i = m\_array[i]$ specifies the number of rows of $B$. The value of $m_i$ must be at least zero.

$n\_array$

Array of size $group\_count$. For group $i$, $0 \leq i < group\_count$, $n_i = n\_array[i]$ specifies the number of columns of $B$. The value of $n_i$ must be at least zero.

$alpha\_array$

Array of size $group\_count$. For group $i$, $0 \leq i < group\_count$, $alpha\_array[i]$ specifies the scalar $alpha_i$.

$a\_array$

Array, size $total\_batch\_count$, of pointers to arrays used to store $A$ matrices.

For group $i$, $0 \leq i < group\_count$, $k$ is $m_i$ when $side_i = \text{CblasLeft}$ and is $n_i$ when $side_i = \text{CblasRight}$ and $a$ is any of the $group\_size[i]$ arrays starting with $a\_array[group\_size[0] + group\_size[1] + \ldots + group\_size(i - 1)]$.

Before entry with $uplo_i = \text{CblasUpper}$, the leading $k$ by $k$ upper triangular part of the array $a$ must contain the upper triangular matrix and the strictly lower triangular part of $a$ is not referenced.

Before entry with $uplo_i = \text{CblasLower}$ lower triangular part of the array $a$ must contain the lower triangular matrix and the strictly upper triangular part of $a$ is not referenced.

When $diag_i = \text{CblasUnit}$, the diagonal elements of $a$ are not referenced either, but are assumed to be unity.

$lda\_array$

Array of size $group\_count$. For group $i$, $0 \leq i < group\_count$, $lda_i = lda\_array[i]$ specifies the leading dimension of $a$ as declared in the calling (sub)program. When $side_i = \text{CblasLeft}$, then $lda_i$ must be at least $\max(1, m_i)$, when $side_i = \text{CblasRight}$, then $lda_i$ must be at least $\max(1, n_i)$.

$b\_array$

Array, size $total\_batch\_count$, of pointers to arrays used to store $B$ matrices.

For group $i$, $0 \leq i < group\_count$, $b$ is any of the $group\_size[i]$ arrays starting with $b\_array[group\_size[0] + group\_size[1] + \ldots + group\_size(i - 1)]$.

For $Layout = \text{CblasColMajor}$: before entry, the leading $m_i$-by-$n_i$ part of the array $b$ must contain the matrix $B$.

For $Layout = \text{CblasRowMajor}$: before entry, the leading $n_i$-by-$m_i$ part of the array $b$ must contain the matrix $B$.

$ldb\_array$

Array of size $group\_count$. Specifies the leading dimension of $b$ as declared in the calling (sub)program. When $Layout = \text{CblasColMajor}$, $ldb$ must be at least $\max(1, m)$; otherwise, $ldb$ must be at least $\max(1, n)$.

Array of size $group\_count$. For group $i$, $0 \leq i < group\_count$, $ldb_i = ldb\_array[i]$ specifies the leading dimension of $b$ as declared in the calling (sub)program. When $Layout = \text{CblasColMajor}$, $ldb_i$ must be at least $\max(1, m_i)$; otherwise, $ldb_i$ must be at least $\max(1, n_i)$.
**group_count**

Specifies the number of groups. Must be at least 0.

**group_size**

Array of size group_count. The element group_size[i] specifies the number of matrices in group i. Each element in group_size must be at least 0.

**Output Parameters**

**b_array**

Overwritten by the solution matrix X.

---

**mkl_?imatcopy**

*Performs scaling and in-place transposition/copying of matrices.*

**Syntax**

```c
void mkl_simatcopy (const char ordering, const char trans, size_t rows, size_t cols, const float alpha, float * AB, size_t lda, size_t ldb);
void mkl_dimatcopy (const char ordering, const char trans, size_t rows, size_t cols, const double alpha, double * AB, size_t lda, size_t ldb);
void mkl_cimatcopy (const char ordering, const char trans, size_t rows, size_t cols, const MKL_Complex8 alpha, MKL_Complex8 * AB, size_t lda, size_t ldb);
void mkl_zimatcopy (const char ordering, const char trans, size_t rows, size_t cols, const MKL_Complex16 alpha, MKL_Complex16 * AB, size_t lda, size_t ldb);
```

**Include Files**

- mkl.h

**Description**

The mkl_?imatcopy routine performs scaling and in-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

\[ AB := \alpha \cdot \text{op}(AB). \]

**NOTE**

Different arrays must not overlap.

**Input Parameters**

**ordering**

Ordering of the matrix storage.

If ordering = 'R' or 'r', the ordering is row-major.

If ordering = 'C' or 'c', the ordering is column-major.

**trans**

Parameter that specifies the operation type.

If trans = 'N' or 'n', \( \text{op}(AB) = AB \) and the matrix AB is assumed unchanged on input.
If $\text{trans} = 'T'$ or 't', it is assumed that $AB$ should be transposed.

If $\text{trans} = 'C'$ or 'c', it is assumed that $AB$ should be conjugate transposed.

If $\text{trans} = 'R'$ or 'r', it is assumed that $AB$ should be only conjugated.

If the data is real, then $\text{trans} = 'R'$ is the same as $\text{trans} = 'N'$, and $\text{trans} = 'C'$ is the same as $\text{trans} = 'T'$.

**rows**

The number of rows in matrix $AB$ before the transpose operation.

**cols**

The number of columns in matrix $AB$ before the transpose operation.

**ab**

Array.

**alpha**

This parameter scales the input matrix by $\alpha$.

**lda**

Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix; measured in the number of elements.

This parameter must be at least $\text{rows}$ if $\text{ordering} = 'C'$ or 'c', and $\max(1, \text{cols})$ otherwise.

**ldb**

Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix; measured in the number of elements.

To determine the minimum value of $\text{ldb}$ on output, consider the following guideline:

If $\text{ordering} = 'C'$ or 'c', then

- If $\text{trans} = 'T'$ or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{cols})$
- If $\text{trans} = 'N'$ or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{rows})$

If $\text{ordering} = 'R'$ or 'r', then

- If $\text{trans} = 'T'$ or 't' or 'C' or 'c', this parameter must be at least $\max(1, \text{rows})$
- If $\text{trans} = 'N'$ or 'n' or 'R' or 'r', this parameter must be at least $\max(1, \text{cols})$

**Output Parameters**

**ab**

Array.

Contains the matrix $AB$.

**Interfaces**

**mkl_?omatcopy**

*Performs scaling and out-place transposition/copying of matrices.*
Syntax

```c
void mkl_somatcopy (char ordering, char trans, size_t rows, size_t cols, const float alpha, const float * A, size_t lda, float * B, size_t ldb);
void mkl_domatcopy (char ordering, char trans, size_t rows, size_t cols, const double alpha, const double * A, size_t lda, double * B, size_t ldb);
void mkl_comatcopy (char ordering, char trans, size_t rows, size_t cols, const MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, MKL_Complex8 * B, size_t ldb);
void mkl_zomatcopy (char ordering, char trans, size_t rows, size_t cols, const MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, MKL_Complex16 * B, size_t ldb);
```

Include Files

- mkl.h

Description

The mkl_?omatcopy routine performs scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

\[ B := \alpha \cdot \text{op}(A) \]

**NOTE**

Different arrays must not overlap.

Input Parameters

- **ordering**
  
  Ordering of the matrix storage.
  
  If `ordering = 'R'` or `r'`, the ordering is row-major.
  
  If `ordering = 'C'` or `c'`, the ordering is column-major.

- **trans**
  
  Parameter that specifies the operation type.
  
  If `trans = 'N'` or `n'`, `\text{op}(A) = A` and the matrix `A` is assumed unchanged on input.
  
  If `trans = 'T'` or `t'`, it is assumed that `A` should be transposed.
  
  If `trans = 'C'` or `c'`, it is assumed that `A` should be conjugate transposed.
  
  If `trans = 'R'` or `r'`, it is assumed that `A` should be only conjugated.
  
  If the data is real, then `trans = 'R'` is the same as `trans = 'N'`, and `trans = 'C'` is the same as `trans = 'T'`.

- **rows**
  
  The number of rows in matrix `B` (the source matrix).

- **cols**
  
  The number of columns in matrix `B` (the source matrix).

- **alpha**
  
  This parameter scales the input matrix by `alpha`.

- **a**
  
  Array.
lda

If ordering = 'R' or 'r', lda represents the number of elements in array a between adjacent rows of matrix A; lda must be at least equal to the number of columns of matrix A.

If ordering = 'C' or 'c', lda represents the number of elements in array a between adjacent columns of matrix A; lda must be at least equal to the number of row in matrix A.

b

If ordering = 'R' or 'r', ldb represents the number of elements in array b between adjacent rows of matrix B.

• If trans = 'T' or 't' or 'C' or 'c', ldb must be at least equal to rows.
• If trans = 'N' or 'n' or 'R' or 'r', ldb must be at least equal to cols.

If ordering = 'C' or 'c', ldb represents the number of elements in array b between adjacent columns of matrix B.

• If trans = 'T' or 't' or 'C' or 'c', ldb must be at least equal to cols.
• If trans = 'N' or 'n' or 'R' or 'r', ldb must be at least equal to rows.

Output Parameters

b

Array, size at least m.

Contains the destination matrix.

Interfaces

mkl_somatcopy2
Performs two-strided scaling and out-of-place transposition/copying of matrices.

Syntax

void mkl_somatcopy2 (char ordering, char trans, size_t rows, size_t cols, const float * A, size_t lda, size_t stridea, float * B, size_t ldb, size_t strideb);

void mkl_domatcopy2 (char ordering, char trans, size_t rows, size_t cols, const double * A, size_t lda, size_t stridea, double * B, size_t ldb, size_t strideb);

void mkl_comatcopy2 (char ordering, char trans, size_t rows, size_t cols, const MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, size_t stridea, MKL_Complex8 * B, size_t ldb, size_t strideb);

void mkl_zomatcopy2 (char ordering, char trans, size_t rows, size_t cols, const MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, size_t stridea, MKL_Complex16 * B, size_t ldb, size_t strideb);
Include Files
- mkl.h

Description

The mkl_omatcopy2 routine performs two-strided scaling and out-of-place transposition/copying of matrices. A transposition operation can be a normal matrix copy, a transposition, a conjugate transposition, or just a conjugation. The operation is defined as follows:

\[ B := \alpha \cdot \text{op}(A) \]

Normally, matrices in the BLAS or LAPACK are specified by a single stride index. For instance, in the column-major order, \( A(2,1) \) is stored in memory one element away from \( A(1,1) \), but \( A(1,2) \) is a leading dimension away. The leading dimension in this case is at least the number of rows of the source matrix. If a matrix has two strides, then both \( A(2,1) \) and \( A(1,2) \) may be an arbitrary distance from \( A(1,1) \).

**NOTE**
Different arrays must not overlap.

Input Parameters

- **ordering**
  Ordering of the matrix storage.
  - If \( \text{ordering} = 'R' \) or \( 'r' \), the ordering is row-major.
  - If \( \text{ordering} = 'C' \) or \( 'c' \), the ordering is column-major.

- **trans**
  Parameter that specifies the operation type.
  - If \( \text{trans} = 'N' \) or \( 'n' \), \( \text{op}(A) = A \) and the matrix \( A \) is assumed unchanged on input.
  - If \( \text{trans} = 'T' \) or \( 't' \), it is assumed that \( A \) should be transposed.
  - If \( \text{trans} = 'C' \) or \( 'c' \), it is assumed that \( A \) should be conjugate transposed.
  - If \( \text{trans} = 'R' \) or \( 'r' \), it is assumed that \( A \) should be only conjugated.
  - If the data is real, then \( \text{trans} = 'R' \) is the same as \( \text{trans} = 'N' \), and \( \text{trans} = 'C' \) is the same as \( \text{trans} = 'T' \).

- **rows**
  The number of rows in matrix \( B \) (the source matrix).

- **cols**
  The number of columns in matrix \( B \) (the source matrix).

- **alpha**
  This parameter scales the input matrix by \( \alpha \).

- **a**
  Array.

- **lda**
  If \( \text{ordering} = 'R' \) or \( 'r' \), \( lda \) represents the number of elements in array \( a \) between adjacent rows of matrix \( A \); \( lda \) must be at least equal to the number of columns of matrix \( A \).
  - If \( \text{ordering} = 'C' \) or \( 'c' \), \( lda \) represents the number of elements in array \( a \) between adjacent columns of matrix \( A \); \( lda \) must be at least 1 and not more than the number of columns in matrix \( A \).
If `ordering = 'R'` or `r'`, `stridea` represents the number of elements in array `a` between adjacent columns of matrix `A`. `stridea` must be at least 1 and not more than the number of columns in matrix `A`.

If `ordering = 'C'` or `c'`, `stridea` represents the number of elements in array `a` between adjacent rows of matrix `A`. `stridea` must be at least equal to the number of columns in matrix `A`.

Array `b`.

If `ordering = 'R'` or `r'`, `ldb` represents the number of elements in array `b` between adjacent rows of matrix `B`.

- If `trans = 'T'` or `t'` or `C'` or `c'`, `ldb` must be at least equal to `rows/strideb`.
- If `trans = 'N'` or `n'` or `R'` or `r'`, `ldb` must be at least equal to `cols/strideb`.

If `ordering = 'C'` or `c'`, `ldb` represents the number of elements in array `b` between adjacent columns of matrix `B`.

- If `trans = 'T'` or `t'` or `C'` or `c'`, `ldb` must be at least 1 and not more than `rows/strideb`.
- If `trans = 'N'` or `n'` or `R'` or `r'`, `ldb` must be at least 1 and not more than `cols/strideb`.

Array, size at least `m`.

Contains the destination matrix.

Output Parameters

Parameters

`b` Array, size at least `m`.

Contains the destination matrix.

Interfaces

`mkl_?omatadd`

Scales and sums two matrices including in addition to performing out-of-place transposition operations.
Syntax

void mkl_somatadd (char ordering, char transa, char transb, size_t m, size_t n, const float alpha, const float * A, size_t lda, const float beta, const float * B, size_t ldb, float * C, size_t ldc);

void mkl_domatadd (char ordering, char transa, char transb, size_t m, size_t n, const double alpha, const double * A, size_t lda, const double beta, const double * B, size_t ldb, double * C, size_t ldc);

void mkl_comatadd (char ordering, char transa, char transb, size_t m, size_t n, const MKL_Complex8 alpha, const MKL_Complex8 * A, size_t lda, const MKL_Complex8 beta, const MKL_Complex8 * B, size_t ldb, MKL_Complex8 * C, size_t ldc);

void mkl_zomatadd (char ordering, char transa, char transb, size_t m, size_t n, const MKL_Complex16 alpha, const MKL_Complex16 * A, size_t lda, const MKL_Complex16 beta, const MKL_Complex16 * B, size_t ldb, MKL_Complex16 * C, size_t ldc);

Include Files

• mkl.h

Description

The mkl_omatadd routine scales and adds two matrices, as well as performing out-of-place transposition operations. A transposition operation can be no operation, a transposition, a conjugate transposition, or a conjugation (without transposition). The following out-of-place memory movement is done:

\[ C := \alpha \cdot \text{op}(A) + \beta \cdot \text{op}(B) \]

where the \( \text{op}(A) \) and \( \text{op}(B) \) operations are transpose, conjugate-transpose, conjugate (no transpose), or no transpose, depending on the values of \( \text{transa} \) and \( \text{transb} \). If no transposition of the source matrices is required, \( m \) is the number of rows and \( n \) is the number of columns in the source matrices \( A \) and \( B \). In this case, the output matrix \( C \) is \( m \)-by-\( n \).

\[ \text{NOTE} \]
Note that different arrays must not overlap.

Input Parameters

ordering

Ordering of the matrix storage.
- If \( \text{ordering} = 'R' \) or \( 'r' \), the ordering is row-major.
- If \( \text{ordering} = 'C' \) or \( 'c' \), the ordering is column-major.

transa

Parameter that specifies the operation type on matrix \( A \).
- If \( \text{transa} = 'N' \) or \( 'n' \), \( \text{op}(A) = A \) and the matrix \( A \) is assumed unchanged on input.
- If \( \text{transa} = 'T' \) or \( 't' \), it is assumed that \( A \) should be transposed.
- If \( \text{transa} = 'C' \) or \( 'c' \), it is assumed that \( A \) should be conjugate transposed.
- If \( \text{transa} = 'R' \) or \( 'r' \), it is assumed that \( A \) should be conjugated (and not transposed).
If the data is real, then \( \text{transa} = 'R' \) is the same as \( \text{transa} = 'N' \), and \( \text{transa} = 'C' \) is the same as \( \text{transa} = 'T' \).

\( \text{transb} \)

Parameter that specifies the operation type on matrix \( B \).

If \( \text{transb} = 'N' \) or \( 'n' \), \( \text{op}(B) = B \) and the matrix \( B \) is assumed unchanged on input.

If \( \text{transb} = 'T' \) or \( 't' \), it is assumed that \( B \) should be transposed.

If \( \text{transb} = 'C' \) or \( 'c' \), it is assumed that \( B \) should be conjugate transposed.

If \( \text{transb} = 'R' \) or \( 'r' \), it is assumed that \( B \) should be conjugated (and not transposed).

If the data is real, then \( \text{transb} = 'R' \) is the same as \( \text{transb} = 'N' \), and \( \text{transb} = 'C' \) is the same as \( \text{transb} = 'T' \).

\( m \)

The number of matrix rows in \( \text{op}(A) \), \( \text{op}(B) \), and \( C \).

\( n \)

The number of matrix columns in \( \text{op}(A) \), \( \text{op}(B) \), and \( C \).

\( \alpha \)

This parameter scales the input matrix by \( \alpha \).

\( a \)

Array.

\( \text{lda} \)

Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix \( A \); measured in the number of elements.

For \( \text{ordering} = 'C' \) or \( 'c' \): when \( \text{transa} = 'N', 'n', 'R', \) or \( 'r' \), \( \text{lda} \) must be at least \( \max(1, m) \); otherwise \( \text{lda} \) must be \( \max(1, n) \).

For \( \text{ordering} = 'R' \) or \( 'r' \): when \( \text{transa} = 'N', 'n', 'R', \) or \( 'r' \), \( \text{lda} \) must be at least \( \max(1, n) \); otherwise \( \text{lda} \) must be \( \max(1, m) \).

\( \beta \)

This parameter scales the input matrix by \( \beta \).

\( b \)

Array.

\( \text{ldb} \)

Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the source matrix \( B \); measured in the number of elements.

For \( \text{ordering} = 'C' \) or \( 'c' \): when \( \text{transa} = 'N', 'n', 'R', \) or \( 'r' \), \( \text{ldb} \) must be at least \( \max(1, m) \); otherwise \( \text{ldb} \) must be \( \max(1, n) \).

For \( \text{ordering} = 'R' \) or \( 'r' \): when \( \text{transa} = 'N', 'n', 'R', \) or \( 'r' \), \( \text{ldb} \) must be at least \( \max(1, n) \); otherwise \( \text{ldb} \) must be \( \max(1, m) \).

\( \text{ldc} \)

Distance between the first elements in adjacent columns (in the case of the column-major order) or rows (in the case of the row-major order) in the destination matrix \( C \); measured in the number of elements.

If \( \text{ordering} = 'C' \) or \( 'c' \), then \( \text{ldc} \) must be at least \( \max(1, m) \), otherwise \( \text{ldc} \) must be at least \( \max(1, n) \).

**Output Parameters**

\( c \)

Array.
Interfaces

**cblas_?gemm_pack_get_size, cblas_gemm_*_pack_get_size**

*Returns the number of bytes required to store the packed matrix.*

**Syntax**

```
size_t cblas_sgemm_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m, const MKL_INT n, const MKL_INT k)
size_t cblas_dgemm_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m, const MKL_INT n, const MKL_INT k)
size_t cblas_gemm_s8u8s32_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m, const MKL_INT n, const MKL_INT k)
size_t cblas_gemm_s16s16s32_pack_get_size (const CBLAS_IDENTIFIER identifier, const MKL_INT m, const MKL_INT n, const MKL_INT k)
```

**Include Files**

- mkl.h

**Description**

The `cblas_?gemm_pack_get_size` and `cblas_gemm_*_pack_get_size` routines belong to a set of related routines that enable the use of an internal packed storage. Call the `cblas_?gemm_pack_get_size` and `cblas_gemm_*_pack_get_size` routines first to query the size of storage required for a packed matrix structure to be used in subsequent calls. Ultimately, the packed matrix structure is used to compute

\[
C := \alpha \cdot \text{op}(\ A) \cdot \text{op}(\ B) + \beta C
\]

for single and double precision or

\[
C := \alpha \cdot (\text{op}(\ A + A_{\text{offset}})) \cdot (\text{op}(\ B + B_{\text{offset}})) + \beta C + C_{\text{offset}}
\]

for integer type.

where:

- \(\text{op}(X)\) is one of the operations \(\text{op}(X) = X\) or \(\text{op}(X) = X^T\)
- \(\alpha\) and \(\beta\) are scalars,
- \(A, A_{\text{offset}}, B, B_{\text{offset}}, C,\) and \(C_{\text{offset}}\) are matrices
- \(\text{op}(A)\) is an \(m\)-by-\(k\) matrix,
- \(\text{op}(B)\) is a \(k\)-by-\(n\) matrix,
- \(C\) is an \(m\)-by-\(n\) matrix.
- \(A_{\text{offset}}\) is an \(m\)-by-\(k\) matrix.
- \(B_{\text{offset}}\) is an \(k\)-by-\(n\) matrix.
- \(C_{\text{offset}}\) is an \(m\)-by-\(n\) matrix.

**Input Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier</td>
<td>CBLAS_IDENTIFIER</td>
<td>Specifies which matrix is to be packed: If (\text{identifier} = \text{CblasAMatrix}), the size returned is the size required to store matrix (A) in an internal format.</td>
</tr>
</tbody>
</table>
If \( \texttt{identifier} = \texttt{CblasBMatrix} \), the size returned is the size required to store matrix \( B \) in an internal format.

- \( m \) \( \text{MKL_INT} \): Specifies the number of rows of matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

- \( n \) \( \text{MKL_INT} \): Specifies the number of columns of matrix \( \text{op}(B) \) and the number of columns of matrix \( C \). The value of \( n \) must be at least zero.

- \( k \) \( \text{MKL_INT} \): Specifies the number of columns of matrix \( \text{op}(A) \) and the number of rows of matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m )</td>
<td>\text{MKL_INT}</td>
<td>Specifies the number of rows of matrix ( \text{op}(A) ) and of the matrix ( C ). The value of ( m ) must be at least zero.</td>
</tr>
<tr>
<td>( n )</td>
<td>\text{MKL_INT}</td>
<td>Specifies the number of columns of matrix ( \text{op}(B) ) and the number of columns of matrix ( C ). The value of ( n ) must be at least zero.</td>
</tr>
<tr>
<td>( k )</td>
<td>\text{MKL_INT}</td>
<td>Specifies the number of columns of matrix ( \text{op}(A) ) and the number of rows of matrix ( \text{op}(B) ). The value of ( k ) must be at least zero.</td>
</tr>
</tbody>
</table>

**Return Values**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{size} )</td>
<td>\text{size_t}</td>
<td>Returns the size (in bytes) required to store the matrix when packed into the internal format of Intel® MKL.</td>
</tr>
</tbody>
</table>

**Example**

See the following examples in the MKL installation directory to understand the use of these routines:

- \texttt{cblas_sgemm\_pack\_get\_size}: examples\cblas\source\cblas\sgemm\computex.c
- \texttt{cblas_dgemm\_pack\_get\_size}: examples\cblas\source\cblas\dgemm\computex.c
- \texttt{cblas\_gemm\_s8u8s32\_pack\_get\_size}: examples\cblas\source\cblas\sgemm\s8u8s32\computex.c
- \texttt{cblas\_gemm\_s16u16s32\_pack\_get\_size}: examples\cblas\source\cblas\sgemm\s16s16s32\computex.c

**See Also**

- \texttt{cblas\_?gemm\_pack} and \texttt{cblas\_gemm\_\*\_pack}
  - to pack the matrix into a buffer allocated previously.
- \texttt{cblas\_?gemm\_compute} and \texttt{cblas\_gemm\_\*\_compute}
  - to compute a matrix-matrix product with general matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.

**\texttt{cblas\_?gemm\_alloc}**

*Allocates storage for a packed matrix (deprecated).*

**Syntax**

- \( \text{float}^* \texttt{cblas\_sgemm\_alloc} (\text{const} \text{CBLAS\_IDENTIFIER} \texttt{identifier}, \text{const} \text{MKL\_INT} \texttt{m}, \text{const} \text{MKL\_INT} \texttt{n}, \text{const} \text{MKL\_INT} \texttt{k}); \)
- \( \text{double}^* \texttt{cblas\_dgemm\_alloc} (\text{const} \text{CBLAS\_IDENTIFIER} \texttt{identifier}, \text{const} \text{MKL\_INT} \texttt{m}, \text{const} \text{MKL\_INT} \texttt{n}, \text{const} \text{MKL\_INT} \texttt{k}); \)

**Include Files**

- \texttt{mkl\_h}
Description

The `cblas_?gemm_alloc` routine is one of a set of related routines that enable use of an internal packed storage. Call the `cblas_?gemm_alloc` routine first to allocate storage for a packed matrix structure to be used in subsequent calls, ultimately to compute

\[ C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C, \]

where:

- \( \text{op}(X) \) is one of the operations \( \text{op}(X) = X, \text{op}(X) = X^T, \) or \( \text{op}(X) = X^H, \)
- \( \alpha \) and \( \beta \) are scalars,
- \( A, B, \) and \( C \) are matrices:
  - \( \text{op}(A) \) is an \( m \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.

Notice

The `cblas_?gemm_alloc` routine is not supported on Windows* OS for the IA-32 architecture with single dynamic library linking.

Input Parameters

- \textit{identifier} Specifies which matrix is to be packed:
  - If \textit{identifier} = CblasAMatrix, the routine allocates storage to pack matrix \( A. \)
  - If \textit{identifier} = CblasBMatrix, the routine allocates storage to pack matrix \( B. \)
- \textit{m} Specifies the number of rows of matrix \( \text{op}(A) \) and of the matrix \( C. \) The value of \( m \) must be at least zero.
- \textit{n} Specifies the number of columns of matrix \( \text{op}(B) \) and the number of columns of matrix \( C. \) The value of \( n \) must be at least zero.
- \textit{k} Specifies the number of columns of matrix \( \text{op}(A) \) and the number of rows of matrix \( \text{op}(B). \) The value of \( k \) must be at least zero.

Return Values

The function returns the allocated storage.

See Also

- `cblas_?gemm_pack` Performs scaling and packing of the matrix into the previously allocated buffer.
- `cblas_?gemm_compute` Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
- `cblas_?gemm_free` Frees the storage previously allocated for the packed matrix (deprecated).
- `cblas_?gemm` for a detailed description of general matrix multiplication.
**cblas_?gemm_pack**  
Performs scaling and packing of the matrix into the previously allocated buffer.

**Syntax**

```c
void cblas_sgemm_pack(  
    const CBLAS_LAYOUT Layout,  
    const CBLAS_IDENTIFIER identifier,  
    const CBLAS_TRANSPOSE trans,  
    const MKL_INT m,  
    const MKL_INT n,  
    const MKL_INT k,  
    const float alpha,  
    const float *src,  
    const MKL_INT ld,  
    float *dest);
```

```c
void cblas_dgemm_pack(  
    const CBLAS_LAYOUT Layout,  
    const CBLAS_IDENTIFIER identifier,  
    const CBLAS_TRANSPOSE trans,  
    const MKL_INT m,  
    const MKL_INT n,  
    const MKL_INT k,  
    const double alpha,  
    const double *src,  
    const MKL_INT ld,  
    double *dest);
```

**Include Files**

- mkl.h

**Description**

The `cblas_?gemm_pack` routine is one of a set of related routines that enable use of an internal packed storage. Call `cblas_?gemm_pack` after you allocate a buffer whose size is given by `cblas_?gemm_pack_getsize`. The `cblas_?gemm_pack` routine scales the identified matrix by alpha and packs it into the buffer allocated previously.

The `cblas_?gemm_pack` routine performs this operation:

\[ \text{dest} := \text{alpha} \cdot \text{op(src)} \text{ as part of the computation } C := \alpha \cdot \text{op}(A) \cdot \text{op}(B) + \beta \cdot C \]

where:

- `op(X)` is one of the operations \( X \), \( X^T \), or \( X^H \),
- `alpha` and `beta` are scalars,
- `src` is a matrix,
- `A`, `B`, and `C` are matrices
- `op(src)` is an \( m \)-by-\( k \) matrix if `identifier` = `CblasAMatrix`,
- `op(src)` is a \( k \)-by-\( n \) matrix if `identifier` = `CblasBMatrix`,
- `dest` is an internal packed storage buffer.

**NOTE**

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_?gemm_pack` and `cblas_?gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both `A` and `B` matrices, you must use the same number of threads for packing `A` as for packing `B`.

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

- **identifier**
  Specifies which matrix is to be packed:
If \( \text{identifier} = \text{CblasAMatrix} \), the routine allocates storage to pack matrix \( A \).

If \( \text{identifier} = \text{CblasBMatrix} \), the routine allocates storage to pack matrix \( B \).

\( \text{trans} \) Specifies the form of \( \text{op} (\text{src}) \) used in the packing:

If \( \text{trans} = \text{CblasNoTrans} \) \( \text{op} (\text{src}) = \text{src} \).

If \( \text{trans} = \text{CblasTrans} \) \( \text{op} (\text{src}) = \text{src}^T \).

If \( \text{trans} = \text{CblasConjTrans} \) \( \text{op} (\text{src}) = \text{src}^H \).

\( m \) Specifies the number of rows of the matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

\( n \) Specifies the number of columns of the matrix \( \text{op}(B) \) and the number of columns of the matrix \( C \). The value of \( n \) must be at least zero.

\( k \) Specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

\( \alpha \) Specifies the scalar \( \alpha \).

\( \text{src} \) Array:

<table>
<thead>
<tr>
<th>identifier = CblasAMatrix</th>
<th>identifier = CblasBMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{trans} = \text{CblasNoTrans} )</td>
<td>( \text{trans} = \text{CblasNoTrans} )</td>
</tr>
<tr>
<td>( \text{trans} = \text{CblasTrans or trans = CblasConjTrans} )</td>
<td>( \text{trans} = \text{CblasTrans or trans = CblasConjTrans} )</td>
</tr>
<tr>
<td><strong>Layout</strong> = CblasColMajor</td>
<td><strong>Layout</strong> = CblasColMajor</td>
</tr>
<tr>
<td>Size ( ld * k ). Before entry, the leading ( m )-by-( k ) part of the array ( \text{src} ) must contain the matrix ( A ).</td>
<td>Size ( ld * m ). Before entry, the leading ( k )-by-( m ) part of the array ( \text{src} ) must contain the matrix ( A ).</td>
</tr>
<tr>
<td>Before entry, the leading ( n )-by-( k ) part of the array ( \text{src} ) must contain the matrix ( B ).</td>
<td>Before entry, the leading ( n )-by-( k ) part of the array ( \text{src} ) must contain the matrix ( B ).</td>
</tr>
</tbody>
</table>

<p>| Layout = CblasRowMajor | Size ( ld * k ). Before entry, the leading ( n )-by-( k ) part of the array ( \text{src} ) must contain the matrix ( B ). |
| Size ( ld * m ). Before entry, the leading ( k )-by-( m ) part of ( \text{src} ) must contain the matrix ( A ). | Size ( ld * k ). Before entry, the leading ( n )-by-( k ) part of ( \text{src} ) must contain the matrix ( B ). |
| <strong>Size</strong> ( ld * n ). Before entry, the leading ( k )-by-( n ) part of ( \text{src} ) must contain the matrix ( B ). | <strong>Size</strong> ( ld * n ). Before entry, the leading ( k )-by-( n ) part of ( \text{src} ) must contain the matrix ( B ). |</p>
<table>
<thead>
<tr>
<th>identifier = CblasAMatrix</th>
<th>identifier = CblasBMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>trans</strong> = CblasNoTrans</td>
<td><strong>trans</strong> = CblasNoTrans</td>
</tr>
<tr>
<td>or trans = CblasTrans</td>
<td>or trans = CblasTrans</td>
</tr>
<tr>
<td>or trans = CblasConjTrans</td>
<td>or trans = CblasConjTrans</td>
</tr>
<tr>
<td>of the array src must contain the matrix A.</td>
<td>of the array src must contain the matrix A.</td>
</tr>
</tbody>
</table>

Specifies the leading dimension of src as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>identifier = CblasAMatrix</th>
<th>identifier = CblasBMatrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>trans</strong> = CblasNoTrans</td>
<td></td>
</tr>
<tr>
<td>or trans = CblasTrans</td>
<td></td>
</tr>
<tr>
<td>or trans = CblasConjTrans</td>
<td></td>
</tr>
<tr>
<td>Layout = CblasColMajor</td>
<td></td>
</tr>
<tr>
<td>ld must be at least max(1, m).</td>
<td>ld must be at least max(1, k).</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td></td>
</tr>
<tr>
<td>ld must be at least max(1, k).</td>
<td>ld must be at least max(1, m).</td>
</tr>
</tbody>
</table>

Scaled and packed internal storage buffer.

**Output Parameters**

dest

Overwritten by the matrix alpha*op(src).

**See Also**

- cblas_gemm_pack_get_size: Returns the number of bytes required to store the packed matrix.
- cblas_gemm_compute: Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
cblas_gemm
for a detailed description of general matrix multiplication.

cblas_gemm_*_pack
Pack the matrix into the buffer allocated previously.

Syntax
void cblas_gemm_s8u8s32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT k, const void *src, const MKL_INT ld, void *dest);
void cblas_gemm_s16s16s32_pack (const CBLAS_LAYOUT Layout, const CBLAS_IDENTIFIER identifier, const CBLAS_TRANSPOSE trans, const MKL_INT m, const MKL_INT n, const MKL_INT k, const MKL_INT16 *src, const MKL_INT ld, MKL_INT16 *dest);

Include Files
• mkl.h

Description
The cblas_gemm_*_pack routine is one of a set of related routines that enable the use of an internal packed storage. Call cblas_gemm_*_pack after you allocate a buffer whose size is given by cblas_gemm_*_pack_get_size. The cblas_gemm_*_pack routine packs the identified matrix into the buffer allocated previously.

The cblas_gemm_*_pack routine performs this operation:
\[ \text{dest} := \text{op} (\text{src}) \text{ as part of the computation } C := \alpha \cdot (\text{op} (A) + A_{\text{offset}}) \cdot (\text{op} (B) + B_{\text{offset}}) + \beta C + C_{\text{offset}} \]

where:
- \( \text{op}(X) \) is one of the operations \( \text{op}(X) = X \) or \( \text{op}(X) = X^T \)
- \( \alpha \) and \( \beta \) are scalars,
- \( \text{src} \) is a matrix,
- \( A, A_{\text{offset}}, B, B_{\text{offset}}, C, \text{and } C_{\text{offset}} \) are matrices
- \( \text{op}(\text{src}) \) is a m-by-k matrix if \( \text{identifier} = \text{CblasAMatrix} \),
- \( \text{op}(\text{src}) \) is a k-by-n matrix if \( \text{identifier} = \text{CblasBMatrix} \),
- \( \text{dest} \) is the buffer previously allocated to store the matrix packed into an internal format
- \( A_{\text{offset}} \) is an m-by-k matrix.
- \( B_{\text{offset}} \) is an k-by-n matrix.
- \( C_{\text{offset}} \) is an m-by-n matrix.

NOTE
You must use the same value of the Layout parameter for the entire sequence of related cblas_gemm_*_pack and cblas_gemm_*_compute calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both \( A \) and \( B \) matrices, you must use the same number of threads for packing \( A \) as for packing \( B \).
Input Parameters

CBLAS_LAYOUT
Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

identifier
CBLAS_IDENTIFIER
Specifies which matrix is to be packed:
If identifier = CblasAMatrix, the A matrix is packed.
If identifier = CblasBMatrix, the B matrix is packed.

trans
CBLAS_TRANSPOSE
Specifies the form of \( \text{op}(\text{src}) \) used in the packing:
If trans = CblasNoTrans \( \text{op}(\text{src}) = \text{src} \).
If trans = CblasTrans \( \text{op}(\text{src}) = \text{src}^T \).

m
MKL_INT
Specifies the number of rows of matrix \( \text{op}(A) \) and of the matrix C. The value of \( m \) must be at least zero.

n
MKL_INT
Specifies the number of columns of matrix \( \text{op}(B) \) and the number of columns of matrix C. The value of \( n \) must be at least zero.

k
MKL_INT
Specifies the number of columns of matrix \( \text{op}(A) \) and the number of rows of matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

src
void* for cblas_gemm_s8u8s32_pack and MKL_INT16* for cblas_gemm_s16s16s32_pack

<table>
<thead>
<tr>
<th>( \text{identifier} = \text{CblasAMatrix} )</th>
<th>( \text{identifier} = \text{CblasBMatrix} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{trans} = \text{CblasNoTrans} )</td>
<td>( \text{trans} = \text{CblasNoTrans} )</td>
</tr>
<tr>
<td>( \text{Size ld} \times k ). Before entry, the leading ( m )-by- ( k ) part of the array ( \text{src} ) must contain the matrix A.</td>
<td>( \text{Size ld} \times m ). Before entry, the leading ( k )-by- ( m ) part of the array ( \text{src} ) must contain the matrix A.</td>
</tr>
<tr>
<td>identifier = CblasAMatrix</td>
<td>identifier = CblasBMatrix</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td>trans = CblasNoTrans</td>
<td>trans = CblasNoTrans</td>
</tr>
<tr>
<td>For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit signed integer.</td>
<td>must be an 8-bit unsigned integer.</td>
</tr>
<tr>
<td>For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit signed integer.</td>
<td>must be an 8-bit unsigned integer.</td>
</tr>
<tr>
<td>Layout = CblasRowMajor</td>
<td>Size ld*m.</td>
</tr>
<tr>
<td>Before entry, the leading (k)-by-(m) part of the array src must contain the matrix (A). For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit unsigned integer.</td>
<td>Size ld*k. Before entry, the leading (n)-by-(k) part of the array src must contain the matrix (B). For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit signed integer.</td>
</tr>
<tr>
<td>Size ld*k. Before entry, the leading (n)-by-(k) part of the array src must contain the matrix (B). For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit signed integer.</td>
<td>Size ld*n. Before entry, the leading (k)-by-(n) part of the array src must contain the matrix (B). For cblas_gemm_s8u8s32_pack the element in src array must be an 8-bit signed integer.</td>
</tr>
</tbody>
</table>

MKL_INT specifies the leading dimension of src as declared in the calling (sub)program.
**Output Parameters**

- `dest` `void*` for `cblas_gemm_s8u8s32_pack` and `MKL_INT16*` for `cblas_gemm_s16s16s32_pack`

  Overwritten by the matrix `op(src)` stored in a format internal to Intel® MKL.

**Example**

See the following examples in the MKL installation directory to understand the use of these routines:

- `cblas_gemm_s8u8s32_pack`: examples\blas\source\cblas_gemm_s8u8s32_computex.c
- `cblas_gemm_s16s16s32_pack`: examples\blas\source\cblas_gemm_s16s16s32_computex.c

**Application Notes**

When using `cblas_gemm_s8u8s32_pack` with row-major layout, the data types of \(A\) and \(B\) must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix \(A\) and an 8-bit signed integer array for matrix \(B\).

**See Also**

- `cblas_gemm_*_pack_get_size`
  - to return the number of bytes needed to store the packed matrix.
- `cblas_gemm_*_compute`
  - to compute a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and add the result to a scalar-matrix product.
**cblas_gemm_compute**

Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.

**Syntax**

```c
void cblas_sgemm_compute(
    const CBLAS_LAYOUT Layout,
    const MKL_INT transa,
    const MKL_INT transb,
    const MKL_INT m,
    const MKL_INT n,
    const MKL_INT k,
    const float *a,
    const MKL_INT lda,
    const float *b,
    const MKL_INT ldb,
    const float beta,
    float *c,
    const MKL_INT ldc);

void cblas_dgemm_compute(
    const CBLAS_LAYOUT Layout,
    const MKL_INT transa,
    const MKL_INT transb,
    const MKL_INT m,
    const MKL_INT n,
    const MKL_INT k,
    const double *a,
    const MKL_INT lda,
    const double *b,
    const MKL_INT ldb,
    const double beta,
    double *c,
    const MKL_INT ldc);
```

**Include Files**

- mkl.h

**Description**

The `cblas_gemm_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_gemm_pack` call `cblas_gemm_compute` to compute

\[ C := \text{op}(A) \ast \text{op}(B) + \beta \ast C, \]

where:

- \( \text{op}(X) \) is one of the operations \( \text{op}(X) = X, \text{op}(X) = X^T, \) or \( \text{op}(X) = X^H, \)
- \( \beta \) is a scalar,
- \( A, B, \) and \( C \) are matrices:
  - \( \text{op}(A) \) is an \( m \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.

**NOTE**

You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_gemm_pack` and `cblas_gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If packing for both \( A \) and \( B \) matrices, you must use the same number of threads for packing \( A \) as for packing \( B \).

**Input Parameters**

- **Layout**
  Specifies whether two-dimensional array storage is row-major (`CblasRowMajor`) or column-major (`CblasColMajor`).

- **transa**
  Specifies the form of \( \text{op}(A) \) used in the matrix multiplication, one of the `CBLAS_TRANSPOSE` or `CBLAS_STORAGE` enumerated types:

  - If `transa = CblasNoTrans` \( \text{op}(A) = A \).
If \( \text{transa} = \text{CblasTrans} \) \( \text{op}(A) = A^T \).

If \( \text{transa} = \text{CblasConjTrans} \) \( \text{op}(A) = A^H \).

If \( \text{transa} = \text{CblasPacked} \) the matrix in array \( a \) is packed and \( lda \) is ignored.

\( \text{transb} \) specifies the form of \( \text{op}(B) \) used in the matrix multiplication, one of the \text{CUBLAS_TRANSPOSE} or \text{CUBLAS_STORAGE} enumerated types:

If \( \text{transb} = \text{CblasNoTrans} \) \( \text{op}(B) = B \).

If \( \text{transb} = \text{CblasTrans} \) \( \text{op}(B) = B^T \).

If \( \text{transb} = \text{CblasConjTrans} \) \( \text{op}(B) = B^H \).

If \( \text{transb} = \text{CblasPacked} \) the matrix in array \( b \) is packed and \( ldb \) is ignored.

\( m \) specifies the number of rows of the matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

\( n \) specifies the number of columns of the matrix \( \text{op}(B) \) and the number of columns of the matrix \( C \). The value of \( n \) must be at least zero.

\( k \) specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

Array:

<table>
<thead>
<tr>
<th>( \text{transa} )</th>
<th>( \text{transa} )</th>
<th>( \text{transa} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( = \text{CblasNoTrans} )</td>
<td>( = \text{CblasTrans or} )</td>
<td>( = \text{CblasPacked} )</td>
</tr>
<tr>
<td>( = \text{CblasTrans} )</td>
<td>( = \text{CblasConjTrans} )</td>
<td></td>
</tr>
</tbody>
</table>

| Layout = \text{CblasColMajor} | Size \( \text{lda} \) \( \times k \). Before entry, the leading \( m \)-by-\( k \) part of the array \( a \) must contain the matrix \( A \). | Size \( \text{lda} \) \( \times m \). Before entry, the leading \( k \)-by-\( m \) part of the array \( a \) must contain the matrix \( A \). | Stored in internal packed format. |
| Layout = \text{CblasRowMajor} | Size \( \text{lda} \) \( \times m \). Before entry, the leading \( k \)-by-\( m \) part of the array \( a \) must contain the matrix \( A \). | Size \( \text{lda} \) \( \times k \). Before entry, the leading \( m \)-by-\( k \) part of the array \( a \) must contain the matrix \( A \). | Stored in internal packed format. |

\( lda \) specifies the leading dimension of \( a \) as declared in the calling (sub)program.
### Parameters

<table>
<thead>
<tr>
<th>Layout</th>
<th>$lda$ must be at least $\max(1, m)$</th>
<th>$lda$ must be at least $\max(1, k)$</th>
<th>$lda$ is ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasColMajor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout</th>
<th>$lda$ must be at least $\max(1, k)$</th>
<th>$lda$ must be at least $\max(1, m)$</th>
<th>$lda$ is ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CblasRowMajor</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### $b$

Array:

<table>
<thead>
<tr>
<th>transb = CblasNoTrans</th>
<th>transb = CblasTrans or transb = CblasConjTrans</th>
<th>transb = CblasPacked</th>
</tr>
</thead>
<tbody>
<tr>
<td>size $ldb \cdot n$.</td>
<td>Before entry, the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$.</td>
<td>Stored in internal packed format.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>transb = CblasPacked</th>
<th>Size $ldb \cdot k$. Before entry, the leading $n$-by-$k$ part of the array $b$ must contain the matrix $B$.</th>
<th>Stored in internal packed format.</th>
</tr>
</thead>
</table>

### $c$

Array:

<table>
<thead>
<tr>
<th>transb = CblasNoTrans</th>
<th>transb = CblasTrans or transb = CblasConjTrans</th>
<th>transb = CblasPacked</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ldb$ must be at least $\max(1, k)$.</td>
<td>$ldb$ must be at least $\max(1, n)$.</td>
<td>$ldb$ is ignored.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$ldb$ must be at least $\max(1, n)$</th>
<th>$ldb$ must be at least $\max(1, k)$.</th>
<th>$ldb$ is ignored.</th>
</tr>
</thead>
</table>

### $beta$

Specifies the scalar $beta$. When $beta$ is equal to zero, then $c$ need not be set on input.
Before entry, the leading m-by-n part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.

Before entry, the leading n-by-m part of the array c must contain the matrix C, except when beta is equal to zero, in which case c need not be set on entry.

Specifies the leading dimension of c as declared in the calling (sub)program.

Overwritten by the m-by-n matrix op(A)*op(B) + beta*C.

See Also
- `cblas_gemm_pack_get_size`: Returns the number of bytes required to store the packed matrix.
- `cblas_gemm_pack`: Performs scaling and packing of the matrix into the previously allocated buffer.
- `cblas_gemm_*_compute`: for a detailed description of general matrix multiplication.

### cblas_gemm_*_compute

**Computes a matrix-matrix product with general integer matrices (where one or both input matrices are stored in a packed data structure) and adds the result to a scalar-matrix product.**

#### Syntax

```c
void cblas_gemm_s8u8s32_compute(const CBLAS_LAYOUT Layout, const MKL_INT transa, const MKL_INT transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const MKL_INT8 oa, const void *b, const MKL_INT ldb, const MKL_INT8 ob, const float beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);

void cblas_gemm_s16s16s32_compute(const CBLAS_LAYOUT Layout, const MKL_INT transa, const MKL_INT transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const MKL_INT16 oa, const MKL_INT16 *b, const MKL_INT ldb, const MKL_INT16 ob, const float beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);
```
Include Files
- mkl.h

Description
The `cblas_gemm_*_compute` routine is one of a set of related routines that enable use of an internal packed storage. After calling `cblas_gemm_*_pack` call `cblas_gemm_*_compute` to compute

\[ C := \alpha(\text{op}(A) + A_{\text{offset}})(\text{op}(B) + B_{\text{offset}}) + \beta C + C_{\text{offset}}, \]

where:
- \( \text{op}(X) \) is either \( \text{op}(X) = X \) or \( \text{op}(X) = X^T \)
- \( \alpha \) and \( \beta \) are scalars
- \( A, B, \) and \( C \) are matrices:
  - \( \text{op}(A) \) is an \( m \)-by-\( k \) matrix,
  - \( \text{op}(B) \) is a \( k \)-by-\( n \) matrix,
  - \( C \) is an \( m \)-by-\( n \) matrix.
- \( A_{\text{offset}} \) is an \( m \)-by-\( k \) matrix with every element equal to the value \( \text{oa} \).
- \( B_{\text{offset}} \) is a \( k \)-by-\( n \) matrix with every element equal to the value \( \text{ob} \).
- \( C_{\text{offset}} \) is an \( m \)-by-\( n \) matrix defined by the \( \text{oc} \) array as described in the description of the \( \text{offsetc} \) parameter.

**NOTE**
You must use the same value of the `Layout` parameter for the entire sequence of related `cblas_*gemm_pack` and `cblas_*gemm_compute` calls.

For best performance, use the same number of threads for packing and for computing.

If you are packing for both \( A \) and \( B \) matrices, you must use the same number of threads for packing \( A \) as for packing \( B \).

Input Parameters

- **Layout**
  - CBLAS_LAYOUT
    - Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major(CblasColMajor).

- **transa**
  - MKL_INT
    - Specifies the form of \( \text{op}(A) \) used in the packing:
      - If \( \text{transa} = \text{CblasNoTrans} \) \( \text{op}(A) = A \).
      - If \( \text{transa} = \text{CblasTrans} \) \( \text{op}(A) = A^T \).
      - If \( \text{transa} = \text{CblasPacked} \) the matrix in array \( a \) is packed into a format internal to Intel® MKL and \( lda \) is ignored.

- **transb**
  - MKL_INT
    - Specifies the form of \( \text{op}(B) \) used in the packing:
      - If \( \text{transb} = \text{CblasNoTrans} \) \( \text{op}(B) = B \).
      - If \( \text{transb} = \text{CblasTrans} \) \( \text{op}(B) = B^T \).
      - If \( \text{transb} = \text{CblasPacked} \) the matrix in array \( b \) is packed into a format internal to Intel® MKL and \( ldb \) is ignored.

- **offsetc**
  - CBLAS_OFFSET
    - Specifies the form of \( C_{\text{offset}} \) used in the matrix multiplication.
If \( \text{offsetc}=\text{CblasFixOffset} \), \( \text{oc} \) has a single element and every element of \( \text{C_offset} \) is equal to this element.

If \( \text{offsetc}=\text{CblasColOffset} \), \( \text{oc} \) has a size of \( m \) and every element of \( \text{C_offset} \) is equal to \( \text{oc} \).

If \( \text{offsetc}=\text{CblasRowOffset} \), \( \text{oc} \) has a size of \( n \) and every element of \( \text{C_offset} \) is equal to \( \text{oc} \).

**\( m \)** MKL_INT Specifies the number of rows of the matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

**\( n \)** MKL_INT Specifies the number of columns of the matrix \( \text{op}(B) \) and the number of columns of the matrix \( C \). The value of \( n \) must be at least zero.

**\( k \)** MKL_INT Specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.

**\( \alpha \)** float Specifies the scalar \( \alpha \).

**\( \text{a} \)** void* for \( \text{gemm}\_s8u8s32\_compute \)

**MKL\_INT16** for \( \text{gemm}\_s16s16s32\_compute \)

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{transa} = \text{CblasNoTrans} )</td>
<td>Array, size ( lda\times k ).</td>
</tr>
<tr>
<td></td>
<td>Before entry, the leading ( m )-by-( k ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
<tr>
<td></td>
<td>For ( \text{cblas}_\text{gemm}_s8u8s32_compute ), the element in the ( a ) array must be an 8-bit signed integer.</td>
</tr>
<tr>
<td>( \text{transa} = \text{CblasTrans} )</td>
<td>Array, size ( lda\times m ).</td>
</tr>
<tr>
<td></td>
<td>Before entry, the leading ( k )-by-( m ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
<tr>
<td></td>
<td>For ( \text{cblas}_\text{gemm}_s8u8s32_compute ), the element in the ( a ) array must be an 8-bit signed integer.</td>
</tr>
<tr>
<td>( \text{transa} = \text{CblasPacked} )</td>
<td>Array of size returned by ( \text{cblas}_\text{gemm}__\text{pack}_\text{get}_\text{size} ) and initialized using ( \text{cblas}_\text{gemm}__\text{pack} ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{transa} = \text{CblasNoTrans} )</td>
<td>Array, size ( lda\times m ).</td>
</tr>
<tr>
<td></td>
<td>Before entry, the leading ( k )-by-( m ) part of the array ( a ) must contain the matrix ( A ).</td>
</tr>
<tr>
<td></td>
<td>For ( \text{cblas}_\text{gemm}_s8u8s32_compute ), the element in the ( a ) array must be an 8-bit unsigned integer.</td>
</tr>
<tr>
<td>( \text{transa} = \text{CblasTrans} )</td>
<td>Array, size ( lda\times k ).</td>
</tr>
</tbody>
</table>

389
Before entry, the leading $m$-by-$k$ part of the array $a$ must contain the matrix $A$.

For `cblas_gemm_s8u8s32_compute`, the element in the $a$ array must be an 8-bit unsigned integer.

Array size returned by `cblas_gemm_*_pack_get_size` and initialized using `cblas_gemm_*_pack`.
### BLAS and Sparse BLAS Routines

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{transa} = \text{CblasNoTrans})</td>
</tr>
<tr>
<td>(\text{transa} = \text{CblasTrans})</td>
</tr>
<tr>
<td>(\text{transa} = \text{CblasPacked})</td>
</tr>
</tbody>
</table>

#### \(ldb\)

\(\text{MKL\_INT}\) Specifies the leading dimension of \(b\) as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>(transb = \text{CblasNoTrans})</th>
<th>(ldb) must be at least (\max(1, k)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{transb} = \text{CblasTrans})</td>
<td>(ldb) must be at least (\max(1, n)).</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layout = CblasRowMajor</th>
<th>(transb = \text{CblasNoTrans})</th>
<th>(ldb) must be at least (\max(1, n)).</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{transb} = \text{CblasTrans})</td>
<td>(ldb) must be at least (\max(1, k)).</td>
<td></td>
</tr>
</tbody>
</table>

#### \(ob\)

\(\text{MKL\_INT8}\) for \(\text{cblas_gemm}_s8u8s32\_compute\)

\(\text{MKL\_INT16}\) for \(\text{cblas_gemm}_s16s16s32\_compute\)

\(\text{Specifies the scalar offset value for the matrix } B.\)

#### \(beta\)

\(\text{float}\)

\(\text{Specifies the scalar } beta.\)

#### \(c\)

\(\text{MKL\_INT32}\)

\(\text{Array:}\)

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>Array, size (ldc\times n). Before entry, the leading (m)-by-(n) part of the array (c) must contain the matrix (C), except when (beta) is equal to zero, in which case (c) need not be set on entry.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{layout} = \text{CblasRowMajor})</td>
<td>Array, size (ldc\times m).</td>
</tr>
</tbody>
</table>
Before entry, the leading \( n \)-by-\( m \) part of the array \( c \) must contain the matrix \( C \), except when \( \beta \) is equal to zero, in which case \( c \) need not be set on entry.

### ldc
MKL_INT Specifies the leading dimension of \( c \) as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>( \text{ldc must be at least max}(1, m) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasRowMajor</td>
<td>( \text{ldc must be at least max}(1, n) )</td>
</tr>
</tbody>
</table>

### oc
MKL_INT32*
Array, size \( \text{len} \). Specifies the scalar offset value for the matrix \( C \).

- If \( \text{offsetc} = \text{CblasFixOffset} \), \( \text{len} \) must be at least 1.
- If \( \text{offsetc} = \text{CblasColOffset} \), \( \text{len} \) must be at least \( \text{max}(1, m) \).
- If \( \text{offsetc} = \text{CblasRowOffset} \), \( \text{len} \) must be at least \( \text{max}(1, n) \).

### Output Parameters
MKL_INT32*
Overwritten by the matrix \( \alpha \cdot (\text{op}(A) + A_{\text{offset}}) \cdot (\text{op}(B) + B_{\text{offset}}) + \beta \cdot C + C_{\text{offset}} \).

### Example
See the following examples in the MKL installation directory to understand the use of these routines:
- cblas_gemm_s8u8s32_compute: examples\cblas\source\cblas_gemm_s8u8s32_computex.c
- cblas_gemm_s16s16s32_compute: examples\cblas\source\cblas_gemm_s16s16s32_computex.c

### Application Notes
You can expand the matrix-matrix product in this manner:

\[
(\text{op}(A) + A_{\text{offset}}) \cdot (\text{op}(B) + B_{\text{offset}}) = \text{op}(A) \cdot \text{op}(B) + \text{op}(A) \cdot B_{\text{offset}} + A_{\text{offset}} \cdot \text{op}(B) + A_{\text{offset}} \cdot B_{\text{offset}}
\]

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the \( C \) matrix are scaled with \( \alpha \) and \( \beta \) floating-point values respectively using double-precision arithmetic. Before storing the results to the output \( c \) array, the floating-point values are rounded to the nearest integers.

In the event of overflow or underflow, the results depend on the architecture. The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

When using cblas_gemm_s8u8s32_compute with row-major layout, the data types of \( A \) and \( B \) must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix \( A \) and an 8-bit signed integer array for matrix \( B \).

### See Also
- cblas_gemm_*_pack_get_size
to return the number of bytes needed to store the packed matrix.
- cblas_gemm_*_pack
to pack the matrix into the buffer allocated previously.

**cblas_?gemm_free**

Frees the storage previously allocated for the packed matrix (deprecated).

**Syntax**

```c
void cblas_sgemm_free (float *dest);
void cblas_dgemm_free (double *dest);
```

**Include Files**

- mkl.h

**Description**

The cblas_?gemm_free routine is one of a set of related routines that enable use of an internal packed storage. Call the cblas_?gemm_free routine last to release storage for the packed matrix structure allocated with cblas_?gemm_alloc.

**Input Parameters**

- `dest`: Previously allocated storage.

**Output Parameters**

- `dest`: The freed buffer.

**See Also**

- cblas_?gemm_alloc: Allocates storage for a packed matrix (deprecated).
- cblas_?gemm_pack: Performs scaling and packing of the matrix into the previously allocated buffer.
- cblas_?gemm_compute: Computes a matrix-matrix product with general matrices where one or both input matrices are stored in a packed data structure and adds the result to a scalar-matrix product.
- cblas_?gemm: for a detailed description of general matrix multiplication.

**cblas_gemm_***

Computes a matrix-matrix product with general integer matrices.

**Syntax**

```c
void cblas_gemm_s8u8s32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float alpha, const void *a, const MKL_INT lda, const MKL_INT8 oa, const void *b, const MKL_INT ldb, const MKL_INT8 ob, const float beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);

void cblas_gemm_s16s16s32 (const CBLAS_LAYOUT Layout, const CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb, const CBLAS_OFFSET offsetc, const MKL_INT m, const MKL_INT n, const MKL_INT k, const float alpha, const MKL_INT16 *a, const MKL_INT lda, const MKL_INT16 oa, const MKL_INT16 *b, const MKL_INT ldb, const MKL_INT16 ob, const float beta, MKL_INT32 *c, const MKL_INT ldc, const MKL_INT32 *oc);
```
Include Files

- mkl.h

Description

The cblas_gemm_* routines compute a scalar-matrix-matrix product and adds the result to a scalar-matrix product. To get the final result, a vector is added to each row or column of the output matrix. The operation is defined as:

\[ C := \alpha \cdot (\text{op}(A) + A\text{\_offset}) \cdot (\text{op}(B) + B\text{\_offset}) + \beta \cdot C + C\text{\_offset} \]

where:

- \( \text{op}(X) \) is either \( \text{op}(X) = X \) or \( \text{op}(X) = X^T \),
- \( A\text{\_offset} \) is an \( m \)-by-\( k \) matrix with every element equal to the value \( a \),
- \( B\text{\_offset} \) is a \( k \)-by-\( n \) matrix with every element equal to the value \( b \),
- \( C\text{\_offset} \) is an \( m \)-by-\( n \) matrix defined by the \( oc \) array as described in the description of the \( offsetc \) parameter,
- \( \alpha \) and \( \beta \) are scalars,
- \( A \) is a matrix such that \( \text{op}(A) \) is \( m \)-by-\( k \),
- \( B \) is a matrix such that \( \text{op}(B) \) is \( k \)-by-\( n \),
- and \( C \) is an \( m \)-by-\( n \) matrix.

Input Parameters

- **Layout**
  Specifies whether two-dimensional array storage is row-major (CblasRowMajor) or column-major (CblasColMajor).

- **transa**
  Specifies the form of \( \text{op}(A) \) used in the matrix multiplication:
  - if \( transa=CblasNoTrans \), then \( \text{op}(A) = A \);
  - if \( transa=CblasTrans \), then \( \text{op}(A) = A^T \).

- **transb**
  Specifies the form of \( \text{op}(B) \) used in the matrix multiplication:
  - if \( transb=CblasNoTrans \), then \( \text{op}(B) = B \);
  - if \( transb=CblasTrans \), then \( \text{op}(B) = B^T \).

- **offsetc**
  Specifies the form of \( C\text{\_offset} \) used in the matrix multiplication.
  - \( offsetc = CblasFixOffset \): \( oc \) has a single element and every element of \( C\text{\_offset} \) is equal to this element.
  - \( offsetc = CblasColOffset \): \( oc \) has a size of \( m \) and every column of \( C\text{\_offset} \) is equal to \( oc \).
  - \( offsetc = CblasRowOffset \): \( oc \) has a size of \( n \) and every row of \( C\text{\_offset} \) is equal to \( oc \).

- **m**
  Specifies the number of rows of the matrix \( \text{op}(A) \) and of the matrix \( C \). The value of \( m \) must be at least zero.

- **n**
  Specifies the number of columns of the matrix \( \text{op}(B) \) and the number of columns of the matrix \( C \). The value of \( n \) must be at least zero.

- **k**
  Specifies the number of columns of the matrix \( \text{op}(A) \) and the number of rows of the matrix \( \text{op}(B) \). The value of \( k \) must be at least zero.
alpha

Specifies the scalar $\alpha$.

<table>
<thead>
<tr>
<th></th>
<th>$\text{transa}=\text{CblasNoTrans}$</th>
<th>$\text{transa}=\text{CblasTrans}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CblasColMajor</td>
<td><strong>Array, size $\text{lda}\times k$</strong></td>
<td><strong>Array, size $\text{lda}\times m$</strong></td>
</tr>
<tr>
<td></td>
<td>Before entry, the leading $m$-by-$k$ part of the array $a$ must contain the matrix $A$ of 8-bit signed integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$.</td>
<td>Before entry, the leading $k$-by-$m$ part of the array $a$ must contain the matrix $A$ of 8-bit signed integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$.</td>
</tr>
</tbody>
</table>

|                |                                     |                                     |
| CblasRowMajor  | **Array, size $\text{lda}\times m$** | **Array, size $\text{lda}\times k$** |
|                | Before entry, the leading $k$-by-$m$ part of the array $a$ must contain the matrix $A$ of 8-bit unsigned integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$. | Before entry, the leading $m$-by-$k$ part of the array $a$ must contain the matrix $A$ of 8-bit unsigned integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$. |

**lda**

Specifies the leading dimension of $a$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th></th>
<th>$\text{transa}=\text{CblasNoTrans}$</th>
<th>$\text{transa}=\text{CblasTrans}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CblasColMajor</td>
<td>$\text{lda}$ must be at least $\max(1, m)$.</td>
<td>$\text{lda}$ must be at least $\max(1, k)$.</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td>$\text{lda}$ must be at least $\max(1, k)$.</td>
<td>$\text{lda}$ must be at least $\max(1, m)$.</td>
</tr>
</tbody>
</table>

**oa**

Specifies the scalar offset value for matrix $A$.

<table>
<thead>
<tr>
<th></th>
<th>$\text{transb}=\text{CblasNoTrans}$</th>
<th>$\text{transb}=\text{CblasTrans}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CblasColMajor</td>
<td><strong>Array, size $\text{ldb}$ by $n$</strong></td>
<td><strong>Array, size $\text{ldb}$ by $k$</strong></td>
</tr>
<tr>
<td></td>
<td>Before entry, the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$ of 8-bit unsigned integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$.</td>
<td>Before entry the leading $n$-by-$k$ part of the array $b$ must contain the matrix $B$ of 8-bit unsigned integers for $\text{cblas_gemm_s8u8s32}$ or 16-bit signed integers for $\text{cblas_gemm_s16s16s32}$.</td>
</tr>
<tr>
<td>CblasRowMajor</td>
<td><strong>Array, size $\text{ldb}$ by $k$</strong></td>
<td><strong>Array, size $\text{ldb}$ by $n$</strong></td>
</tr>
</tbody>
</table>

**b**
Before entry the leading $n$-by-$k$ part of the array $b$ must contain the matrix $B$ of 8-bit signed integers for `cblas_gemm_s8u8s32` or 16-bit signed integers for `cblas_gemm_s16s16s32`. Before entry, the leading $k$-by-$n$ part of the array $b$ must contain the matrix $B$ of 8-bit signed integers for `cblas_gemm_s8u8s32` or 16-bit signed integers for `cblas_gemm_s16s16s32`.

**ldb**

Specifies the leading dimension of $b$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>transb=CblasNoTrans</th>
<th>ldb must be at least max(1, k).</th>
<th>transb=CblasTrans</th>
<th>ldb must be at least max(1, n).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasRowMajor</td>
<td>ldb must be at least max(1, n).</td>
<td>ldb must be at least max(1, k).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**ob**

Specifies the scalar offset value for matrix $B$.

**beta**

Specifies the scalar $beta$. When $beta$ is equal to zero, then $c$ need not be set on input.

**c**

Array, size $ldc$ by $n$. Before entry, the leading $m$-by-$n$ part of the array $c$ must contain the matrix $C$, except when $beta$ is equal to zero, in which case $c$ need not be set on entry.

Array, size $ldc$ by $m$. Before entry, the leading $n$-by-$m$ part of the array $c$ must contain the matrix $C$, except when $beta$ is equal to zero, in which case $c$ need not be set on entry.

**ldc**

Specifies the leading dimension of $c$ as declared in the calling (sub)program.

<table>
<thead>
<tr>
<th>Layout = CblasColMajor</th>
<th>ldc must be at least max(1, m).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layout = CblasRowMajor</td>
<td>ldc must be at least max(1, n).</td>
</tr>
</tbody>
</table>

**oc**

Array, size $len$. Specifies the offset values for matrix $C$.

- If $offsetc = CblasFixOffset$, $len$ must be at least 1.
- If $offsetc = CblasColOffset$, $len$ must be at least max(1, $m$).
- If $offsetc = CblasRowOffset$, $oc$ must be at least max(1, $n$).
Output Parameters

Overwritten by $alpha \cdot (\text{op}(A) + A\_\text{offset}) \cdot (\text{op}(B) + B\_\text{offset}) + beta \cdot C + C\_\text{offset}$.

Example

For examples of routine usage, see the code in the Intel® MKL installation directory:

- cblas_gemm_s8u8s32: examples\cblas\source\cblas_gemm_s8u8s32x.c
- cblas_gemm_s16s16s32: examples\cblas\source\cblas_gemm_s16s16s32x.c

Application Notes

The matrix-matrix product can be expanded:

$$(\text{op}(A) + A\_\text{offset}) \cdot (\text{op}(B) + B\_\text{offset})$$

$$= \text{op}(A) \cdot \text{op}(B) + \text{op}(A) \cdot B\_\text{offset} + A\_\text{offset} \cdot \text{op}(B) + A\_\text{offset} \cdot B\_\text{offset}$$

After computing these four multiplication terms separately, they are summed from left to right. The results from the matrix-matrix product and the $C$ matrix are scaled with $alpha$ and $beta$ floating-point values respectively using double-precision arithmetic. Before storing the results to the output $c$ array, the floating-point values are rounded to the nearest integers. In the event of overflow or underflow, the results depend on the architecture. The results are either unsaturated (wrapped) or saturated to maximum or minimum representable integer values for the data type of the output matrix.

When using cblas_gemm_s8u8s32 with row-major layout, the data types of $A$ and $B$ must be swapped. That is, you must provide an 8-bit unsigned integer array for matrix $A$ and an 8-bit signed integer array for matrix $B$.

Intermediate integer computations in cblas_gemm_s8u8s32 on 64-bit Intel® Advanced Vector Extensions 2 (Intel® AVX2) and Intel® Advanced Vector Extensions 512 (Intel® AVX-512) architectures without Vector Neural Network Instructions (VNNI) extensions can saturate. This is because only 16-bits are available for the accumulation of intermediate results. You can avoid integer saturation by maintaining all integer elements of $A$ or $B$ matrices under 8 bits.
LAPACK Routines

This chapter describes the Intel® Math Kernel Library (Intel® MKL) implementation of routines from the LAPACK package that are used for solving systems of linear equations, linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. The library includes LAPACK routines for both real and complex data. Routines are supported for systems of equations with the following types of matrices:

- General
- Banded
- Symmetric or Hermitian positive-definite (full, packed, and rectangular full packed (RFP) storage)
- Symmetric or Hermitian positive-definite banded
- Symmetric or Hermitian indefinite (both full and packed storage)
- Symmetric or Hermitian indefinite banded
- Triangular (full, packed, and RFP storage)
- Triangular banded
- Tridiagonal
- Diagonally dominant tridiagonal.

**NOTE**
Different arrays used as parameters to Intel® MKL LAPACK routines must not overlap.

**Warning**
LAPACK routines assume that input matrices do not contain IEEE 754 special values such as \(\text{INF}\) or NaN values. Using these special values may cause LAPACK to return unexpected results or become unstable.

Choosing a LAPACK Routine

Intel® MKL offers many LAPACK routines, and many perform similar operations. The Intel® Math Kernel Library LAPACK Function Finding Advisor helps you understand the differences between routines so that you can choose the appropriate routine for your task:


C Interface Conventions for LAPACK Routines

The C interfaces are implemented for most of the Intel® MKL LAPACK driver and computational routines.

**NaN checking in LAPACKE**

NaN checking can affect the performance of an application. By default, it is ON.

See the **Support Functions** section for details on the methods and options to turn NaN check off or back on with LAPACKE:

**Function Prototypes**

Intel® MKL supports four distinct floating-point precisions. Each corresponding prototype looks similar, usually differing only in the data type. C interface LAPACK function names follow the form `<fmt><name>`, where `<fmt>` is:
• LAPACKE_s for float
• LAPACKE_d for double
• LAPACKE_c for lapack_complex_float
• LAPACKE_z for lapack_complex_double

A specific example follows. To solve a system of linear equations with a packed Cholesky-factored Hermitian positive-definite matrix with complex precision, use the following:

```c
lapack_int LAPACKE_cpptrs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, lapack_complex_float* b, lapack_int ldb);
```

**Workspace Arrays**

In contrast to the Fortran interface, the LAPACK C interface omits workspace parameters because workspace is allocated during runtime and released upon completion of the function operation.

If you prefer to allocate workspace arrays yourself, the LAPACK C interface provides alternate interfaces with `work` parameters. The name of the alternate interface is the same as the LAPACK C interface with `_work` appended. For example, the syntax for the singular value decomposition of a real bidiagonal matrix is:

**Fortran:**
```
call sbdsdc ( uplo, compq, n, d, e, u, ldu, vt, ldvt, q, iq,
             work, iwork, info )
```

**C LAPACK interface:**
```
lapack_int LAPACKE_sbdsdc ( int matrix_layout, char uplo, char
                           compq, lapack_int n, float* d, float* e, float* u, lapack_int
                           ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq );
```

**Alternate C LAPACK interface with `work` parameters:**
```
lapack_int LAPACKE_sbdsdc_work( int matrix_layout, char uplo,
                            char compq, lapack_int n, float* d, float* e, float* u,
                            lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int*
                            iq, float* work, lapack_int* iwork );
```

See the `install_dir/include/mkl_lapacke.h` file for the full list of alternative C LAPACK interfaces.

The Intel® MKL Fortran-specific documentation contains details about workspace arrays.

**Mapping Fortran Data Types against C Data Types**

<table>
<thead>
<tr>
<th>FORTRAN</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>lapack_int</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>lapack_logical</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>double</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>lapack_complex_float</td>
</tr>
<tr>
<td>COMPLEX*16.DOUBLE COMPLEX</td>
<td>lapack_complex_double</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>char</td>
</tr>
</tbody>
</table>

**C Type Definitions**

You can find type definitions specific to Intel® MKL such as `MKL_INT`, `MKL_Complex8`, and `MKL_Complex16` in `install_dir/mkl_types.h`. 
C types

```c
#ifndef lapack_int
#define lapack_int MKL_INT
#endif

#ifndef lapack_logical
#define lapack_logical lapack_int
#endif
```

Complex Type Definitions

Complex type for single precision:

```c
#ifndef lapack_complex_float
#define lapack_complex_float MKL_Complex8
#endif
```

Complex type for double precision:

```c
#ifndef lapack_complex_double
#define lapack_complex_double MKL_Complex16
#endif
```

Matrix Layout Definitions

```c
#define LAPACK_ROW_MAJOR  101
#define LAPACK_COL_MAJOR  102
```

See Matrix Layout for LAPACK Routines for an explanation of row-major order and column-major order storage.

Error Code Definitions

```c
#define LAPACK_WORK_MEMORY_ERROR       -1010  /* Failed to allocate memory for a working array */
#define LAPACK_TRANSPOSE_MEMORY_ERROR  -1011  /* Failed to allocate memory for transposed matrix */
```

## Matrix Layout for LAPACK Routines

There are two general methods of storing a two dimensional matrix in linear (one dimensional) memory: column-wise (column major order) or row-wise (row major order). Consider an $M$-by-$N$ matrix $A$:

\[
A = \begin{bmatrix}
    a_{11} & a_{12} & \cdots & a_{1L_1} & a_{1L_1+1} & a_{1L_1+2} & \cdots & a_{1L_1+L-1} & a_{1L_1+L} & \cdots & a_{1N} \\
    a_{21} & a_{22} & \cdots & a_{2L_2} & a_{2L_2+1} & a_{2L_2+2} & \cdots & a_{2L_2+L-1} & a_{2L_2+L} & \cdots & a_{2N} \\
    \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
    a_{b+1L_1} & a_{b+1L_1+1} & a_{b+1L_1+2} & \cdots & a_{b+1L_1+L-1} & a_{b+1L_1+L} & \cdots & a_{b+1L_1+L-1} & a_{b+1L_1+L} & \cdots & a_{b+1N} \\
    a_{b+2L_1} & a_{b+2L_1+1} & a_{b+2L_1+2} & \cdots & a_{b+2L_1+L-1} & a_{b+2L_1+L} & \cdots & a_{b+2L_1+L-1} & a_{b+2L_1+L} & \cdots & a_{b+2N} \\
    \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
    a_{b+K-L_1} & a_{b+K-L_1+1} & a_{b+K-L_1+2} & \cdots & a_{b+K-L_1+L-1} & a_{b+K-L_1+L} & \cdots & a_{b+K-L_1+L-1} & a_{b+K-L_1+L} & \cdots & a_{b+K-N} \\
    a_{b+K} & a_{b+K+1} & \cdots & a_{b+K+K-1} & a_{b+K+K} & a_{b+K+K+1} & \cdots & a_{b+K+K} & a_{b+K+K+1} & \cdots & a_{b+K+K-N} \\
    \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\
    a_{M+1} & a_{M+2} & \cdots & a_{M,L_2} & a_{M,L_2+1} & a_{M,L_2+2} & \cdots & a_{M,L_2+L-1} & a_{M,L_2+L} & \cdots & a_{M,N} \\
\end{bmatrix}
\]
**Column Major Layout**

In column major layout the first index, $i$, of matrix elements $a_{ij}$ changes faster than the second index when accessing sequential memory locations. In other words, for $1 \leq i < M$, if the element $a_{ij}$ is stored in a specific location in memory, the element $a_{i+1,j}$ is stored in the next location, and, for $1 \leq j < N$, the element $a_{M,j}$ is stored in the location previous to element $a_{1,j+1}$. So the matrix elements are located in memory according to this sequence:

$$\{a_{1,1}, a_{2,1}, \ldots, a_{M,1}, a_{1,2}, \ldots, a_{M,2}, \ldots, a_{1,N}, a_{2,N}, \ldots, a_{M,N}\}$$

**Row Major Layout**

In row major layout the second index, $j$, of matrix elements $a_{ij}$ changes faster than the first index when accessing sequential memory locations. In other words, for $1 \leq j < N$, if the element $a_{ij}$ is stored in a specific location in memory, the element $a_{i,j+1}$ is stored in the next location, and, for $1 \leq i < M$, the element $a_{i,N}$ is stored in the location previous to element $a_{i+1,1}$. So the matrix elements are located in memory according to this sequence:

$$\{a_{1,1}, a_{1,2}, \ldots, a_{1,N}, a_{2,1}, a_{2,2}, \ldots, a_{2,N}, \ldots, a_{M,1}, a_{M,2}, \ldots, a_{M,N}\}$$

**Leading Dimension Parameter**

A leading dimension parameter allows use of LAPACK routines on a submatrix of a larger matrix. For example, the submatrix $B$ can be extracted from the original matrix $A$ defined previously:

$$B = \begin{bmatrix}
    a_{b+1,j_0+1} & a_{b+1,j_0+2} & \cdots & a_{b+1,j_0+L-1} & a_{b+1,j_0+L} \\
    a_{b+2,j_0+1} & a_{b+2,j_0+2} & \cdots & a_{b+2,j_0+L-1} & a_{b+2,j_0+L} \\
    \vdots & \vdots & \ddots & \vdots & \vdots \\
    a_{b+K-1,j_0+1} & a_{b+K-1,j_0+2} & \cdots & a_{b+K-1,j_0+L-1} & a_{b+K-1,j_0+L} \\
    a_{b+K,j_0+1} & a_{b+K,j_0+2} & \cdots & a_{b+K,j_0+L-1} & a_{b+K,j_0+L}
\end{bmatrix}$$

$B$ is formed from rows with indices $i_0 + 1$ to $i_0 + K$ and columns $j_0 + 1$ to $j_0 + L$ of matrix $A$. To specify matrix $B$, LAPACK routines require four parameters:

- the number of rows $K$;
- the number of columns $L$;
- a pointer to the start of the array containing elements of $B$;
- the leading dimension of the array containing elements of $B$.

The leading dimension depends on the layout of the matrix:

- **Column major layout**
  
  Leading dimension $ldb=M$, the number of rows of matrix $A$.

  Starting address: offset by $i_0 + j_0*ldb$ from $a_{1,1}$.

  $j_0*M + i_0$

  \[
  \begin{array}{cccc}
  a_{1,1} & \cdots & a_{b+1,j_0+1} & a_{b+2,j_0+1} \\
  \downarrow & & \downarrow & \downarrow \\
  a_{b+1,j_0+2} & \cdots & a_{b+K-1,j_0+1} & a_{b+K,j_0+1} \\
  \end{array}
  \]

- **Row major layout**
Leading dimension \( ldb = N \), the number of columns of matrix \( A \).

Starting address: offset by \( i_0 \times ldb + j_0 \) from \( a_{1,1} \).

Matrix Storage Schemes for LAPACK Routines

LAPACK routines use the following matrix storage schemes:

- Full Storage
- Packed Storage
- Band Storage
- Rectangular Full Packed (RFP) Storage

**Full Storage**

Consider an \( m \)-by-\( n \) matrix \( A \):

\[
A = \begin{pmatrix}
a_{1,1} & a_{1,2} & a_{1,3} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & a_{2,3} & \cdots & a_{2,n} \\
a_{3,1} & a_{3,2} & a_{3,3} & \cdots & a_{3,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{m,1} & a_{m,2} & a_{m,3} & \cdots & a_{m,n}
\end{pmatrix}
\]

It is stored in a one-dimensional array \( a \) of length at least \( lda \times n \) for column major layout or \( m \times lda \) for row major layout. Element \( a_{i,j} \) is stored as array element \( a[k] \) where the mapping of \( k(i, j) \) is defined as

- column major layout: \( k(i, j) = i - 1 + (j - 1) \times lda \)
- row major layout: \( k(i, j) = (i - 1) \times lda + j - 1 \)

**NOTE**

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array used to store an \( m \)-by-\( n \) matrix \( A \) with leading dimension \( lda \) should be greater than or equal to \( \max(1, n \times lda) \) for column major layout and \( \max(1, m \times lda) \) for row major layout.

**NOTE**

Even though the array used to store a matrix is one-dimensional, for simplicity the documentation sometimes refers parts of the array such as rows, columns, upper and lower triangular part, and diagonals. These refer to the parts of the matrix stored within the array. For example, the lower triangle of array \( a \) is defined as the subset of elements \( a[k(i,j)] \) with \( k(i,j) \).
Packed Storage

The packed storage format compactly stores matrix elements when only one part of the matrix, the upper or lower triangle, is necessary to determine all of the elements of the matrix. This is the case when the matrix is upper triangular, lower triangular, symmetric, or Hermitian. For an \( n \times n \) matrix of one of these types, a linear array \( a_p \) of length \( n(n + 1)/2 \) is adequate. Two parameters define the storage scheme: \( \text{matrix\_layout} \), which specifies column major (with the value \( \text{LAPACK\_COL\_MAJOR} \)) or row major (with the value \( \text{LAPACK\_ROW\_MAJOR} \)) matrix layout, and \( \text{uplo} \), which specifies that the upper triangle (with the value ‘U’) or the lower triangle (with the value ‘L’) is stored.

Element \( a_{ij} \) is stored as array element \( a[k] \) where the mapping of \( k(i, j) \) is defined as

<table>
<thead>
<tr>
<th>( \text{uplo} )</th>
<th>( \text{matrix_layout} = \text{LAPACK_COL_MAJOR} )</th>
<th>( \text{matrix_layout} = \text{LAPACK_ROW_MAJOR} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘U’</td>
<td>( k(i, j) = i - 1 + j*(j - 1)/2 )</td>
<td>( k(i, j) = j - 1 + (i - 1)<em>(2</em>n - i)/2 )</td>
</tr>
<tr>
<td>‘L’</td>
<td>( k(i, j) = i - 1 + (j - 1)<em>(2</em>n - j)/2 )</td>
<td>( k(i, j) = j - 1 + i*(i - 1)/2 )</td>
</tr>
</tbody>
</table>

NOTE

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to \( \max(1, nx(n + 1)/2) \).

Band Storage

When the non-zero elements of a matrix are confined to diagonal bands, it is possible to store the elements more efficiently using band storage. For example, consider an \( m \times n \) band matrix \( A \) with \( kl \) subdiagonals and \( ku \) superdiagonals:

\[
A = \begin{pmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,k_u + 1} \\
\vdots & \vdots & \ddots & \vdots \\
a_{k_l + 1,1} & a_{k_l + 1,2} & \cdots & a_{k_l + 1,k_u + 1} \\
a_{k_l + 2,2} & \ddots & \ddots & \ddots \\
\vdots & \ddots & \ddots & \ddots \\
a_{k_l + j,j} & a_{k_l + j,j + 1} & \cdots & a_{k_l + j,k_l + k_u + j}
\end{pmatrix}
\]

This matrix can be stored compactly in a one dimensional array \( ab \). There are two operations involved in storing the matrix: packing the band matrix into matrix \( AB \), and converting the packed matrix to a one-dimensional array.

- Packing the Band Matrix: How the band matrix is packed depends on the matrix layout.
  - Column major layout: matrix \( A \) is packed in an \( ldab \)-by-\( n \) matrix \( AB \) column-wise so that the diagonals of \( A \) become rows of array \( AB \).
Converting the Packed Matrix to a One-Dimensional Array: The packed matrix in column major layout:

\[ AB = \begin{pmatrix}
  a_{1,1} & a_{1,2} & \cdots & a_{1,k_u + 1} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{k_l + 1,1} & a_{k_l + 1,2} & \cdots & a_{k_l + 1, k_l + 1} \\
  a_{k_l + 2,1} & a_{k_l + 2,2} & \cdots & a_{k_l + 2, k_l + 2} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{k_l + k_u,1} & a_{k_l + k_u,2} & \cdots & a_{k_l + k_u, k_l + k_u + 1}
\end{pmatrix} \]

The number of rows of \( AB \cdot ldab \geq kl + ku + 1 \), and the number of columns of \( AB \) is \( n \).

- Row major layout: matrix \( A \) is packed in an \( m \)-by-\( ldab \) matrix \( AB \) row-wise so that the diagonals of \( A \) become columns of \( AB \).

\[ AB = \begin{pmatrix}
  a_{1,1} & a_{1,2} & \cdots & a_{1,k_u + 1} \\
  a_{2,1} & a_{2,2} & \cdots & a_{2,k_u + 2} \\
  a_{3,1} & a_{3,2} & \cdots & a_{3,k_u + 3} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{k_l + 1,1} & a_{k_l + 1,2} & \cdots & a_{k_l + 1, k_l + 1} \\
  a_{k_l + 2,1} & a_{k_l + 2,2} & \cdots & a_{k_l + 2, k_l + 2} \\
  a_{k_l + 3,1} & a_{k_l + 3,2} & \cdots & a_{k_l + 3, k_l + 3} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{k_l + k_u,1} & a_{k_l + k_u,2} & \cdots & a_{k_l + k_u, k_l + k_u + 1}
\end{pmatrix} \]

The number of columns of \( AB \cdot ldab \geq kl + ku + 1 \), and the number of rows of \( AB \) is \( m \).

**NOTE**

For both column major and row major layout, elements of the upper left triangle of \( AB \) are not used. Depending on the relationship of the dimensions \( m, n, kl, \) and \( ku \), the lower right triangle might not be used.

- Converting the Packed Matrix to a One-Dimensional Array: The packed matrix \( AB \) is stored in a linear array \( ab \) as described in **Full Storage**. The size of \( ab \) should be greater than or equal to the total number of elements of matrix \( AB: ldab \cdot n \) for column major layout or \( ldab \cdot m \) for row major layout. The leading dimension of \( ab, ldab \), must be greater than or equal to \( kl + ku + 1 \) (and some routines require it to be even larger).

Element \( a_{ij} \) is stored as array element \( a[k(i, j)] \) where the mapping of \( k(i, j) \) is defined as

- column major layout: \( k(i, j) = i + ku - j + (j - 1) \cdot ldab; 1 \leq j \leq n, \max(1, j - ku) \leq k \leq \min(m, j + kl) \)
- row major layout: \( k(i,j) = j - i + k_l + (i - 1)(k_u + 1), 1 \leq i \leq m, \max(1, i - k_l) \leq j \leq \min(n, i + k_u) \)

**NOTE**

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to \( \max(1, n \cdot ldab) \) for column major layout and \( \max(1, m \cdot ldab) \) for row major layout.
Rectangular Full Packed Storage

A combination of full and packed storage, rectangular full packed storage can be used to store the upper or lower triangle of a matrix which is upper triangular, lower triangular, symmetric, or Hermitian. It offers the storage savings of packed storage plus the efficiency of using full storage Level 3 BLAS and LAPACK routines. Three parameters define the storage scheme: matrix_layout, which specifies column major (with the value LAPACK_COL_MAJOR) or row major (with the value LAPACK_ROW_MAJOR) matrix layout; uplo, which specifies that the upper triangle (with the value 'U') or the lower triangle (with the value 'L') is stored; and transr, which specifies normal (with the value 'N'), transpose (with the value 'T'), or conjugate transpose (with the value 'C') operation on the matrix.

Consider an $N$ by $N$ matrix $A$:

$$ A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & \cdots & a_{0,N-1} \\ a_{1,0} & a_{1,1} & a_{1,2} & \cdots & a_{1,N-1} \\ a_{2,0} & a_{2,1} & a_{2,2} & \cdots & a_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{N-1,0} & a_{N-1,1} & a_{N-1,2} & \cdots & a_{N-1,N-1} \end{pmatrix} $$

The upper or lower triangle of $A$ can be stored in the array $ap$ of length $N^2(N + 1)/2$.

Additionally, define $k$ as the integer part of $N/2$, such that $N=2k$ if $N$ is even, and $N=2k + 1$ if $N$ is odd.

Storing the matrix involves packing the matrix into a rectangular matrix, and then storing the matrix in a one-dimensional array. The size of rectangular matrix $AP$ required for the $N$ by $N$ matrix $A$ is $N + 1$ by $N/2$ for even $N$, and $N$ by $(N + 1)/2$ for odd $N$.

These examples illustrate the rectangular full packed storage method.

- **Upper triangular** - $uplo = 'U'$

  Consider a matrix $A$ with $N = 6$:

  $$ A = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\ a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\ a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} \end{pmatrix} $$

- **Not transposed** - $transr = 'N'$

  The elements of the upper triangle of $A$ can be packed in a matrix with the dimensions $(N + 1)$ by $(N/2) = 7$ by $3$:

  $$ AP = \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \\ a_{3,0} & a_{3,1} & a_{3,2} \\ a_{4,0} & a_{4,1} & a_{4,2} \\ a_{5,0} & a_{5,1} & a_{5,2} \end{pmatrix} $$

- **Transposed or conjugate transposed** - $transr = 'T'$ or $transr = 'C'$


The elements of the upper triangle of $A$ can be packed in a matrix with the dimensions $(N/2)$ by $(N + 1) = 3$ by 7:

$$
AP = \begin{pmatrix}
a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{0,0} & a_{0,1} & a_{0,2} \\
a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4} & a_{1,1} & a_{1,2} \\
a_{0,5} & a_{1,5} & a_{2,5} & a_{3,5} & a_{4,5} & a_{5,5} & a_{2,2}
\end{pmatrix}
$$

Consider a matrix $A$ with $N = 5$:

$$
A = \begin{pmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4}
\end{pmatrix}
$$

- **Not transposed - $\text{transr} = 'N'$**

  The elements of the upper triangle of $A$ can be packed in a matrix with the dimensions $(N)$-by-$(N + 1)/2) = 5$ by 3:

  $$
  AP = \begin{pmatrix}
a_{0,2} & a_{0,3} & a_{0,4} \\
a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,2} & a_{2,3} & a_{2,4} \\
a_{0,0} & a_{3,3} & a_{3,4} \\
a_{0,1} & a_{4,1} & a_{4,4}
  \end{pmatrix}
  $$

- **Transposed or conjugate transposed - $\text{transr} = 'T'$ or $\text{transr} = 'C'$**

  The elements of the upper triangle of $A$ can be packed in a matrix with the dimensions $(N+1)/2)$ by $(N) = 5$ by 3:

  $$
  AP = \begin{pmatrix}
a_{0,2} & a_{1,2} & a_{2,3} & a_{0,0} & a_{0,1} \\
a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & a_{1,1} \\
a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & a_{4,4}
  \end{pmatrix}
  $$

- **Lower triangular - $\text{uplo} = 'L'$**

  Consider a matrix $A$ with $N = 6$:

  $$
  A = \begin{pmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} & a_{0,5} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} \\
a_{5,0} & a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5}
  \end{pmatrix}
  $$

- **Not transposed - $\text{transr} = 'N'$**

  The elements of the lower triangle of $A$ can be packed in a matrix with the dimensions $(N + 1)$-by-$(N/2) = 7$ by 3:
The elements of the lower triangle of $A$ can be packed in a matrix with the dimensions $(N/2) \times (N + 1) = 3 \times 7$:

$$AP = \begin{pmatrix}
a_{3,3} & a_{4,3} & a_{5,3} \\
a_{0,0} & a_{1,4} & a_{5,4} \\
a_{1,0} & a_{1,1} & a_{5,5}
\end{pmatrix}$$

- Transposed or conjugate transposed - $\text{transr} = 'T'$ or $\text{transr} = 'C'$

The elements of the lower triangle of $A$ can be packed in a matrix with the dimensions $(N/2) \times (N + 1) = 3 \times 7$:

$$AP = \begin{pmatrix}
a_{3,3} & a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} & a_{5,0} \\
a_{4,3} & a_{4,4} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} & a_{5,1} \\
a_{5,3} & a_{5,4} & a_{5,5} & a_{2,2} & a_{3,2} & a_{4,2} & a_{5,2}
\end{pmatrix}$$

Consider a matrix $A$ with $N = 5$:

$$A = \begin{pmatrix}
a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} & a_{0,4} \\
a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\
a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\
a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\
a_{4,0} & a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4}
\end{pmatrix}$$

- Not transposed - $\text{transr} = 'N'$

The elements of the lower triangle of $A$ can be packed in a matrix with the dimensions $(N) \times ((N + 1)/2) = 5 \times 3$:

$$AP = \begin{pmatrix}
a_{0,0} & a_{3,3} & a_{4,3} \\
a_{1,0} & a_{1,1} & a_{4,4} \\
a_{2,0} & a_{2,1} & a_{2,2} \\
a_{3,0} & a_{3,1} & a_{3,2} \\
a_{4,0} & a_{4,1} & a_{4,2}
\end{pmatrix}$$

- Transposed or conjugate transposed - $\text{transr} = 'T'$ or $\text{transr} = 'C'$

The elements of the lower triangle of $A$ can be packed in a matrix with the dimensions $((N+1)/2) \times (N) = 5 \times 3$:

$$AP = \begin{pmatrix}
a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & a_{4,0} \\
a_{3,3} & a_{1,1} & a_{2,1} & a_{3,1} & a_{4,1} \\
a_{4,3} & a_{4,4} & a_{2,2} & a_{3,2} & a_{4,2}
\end{pmatrix}$$

The packed matrix $AP$ can be stored using column major layout or row major layout.
NOTE
The `matrix_layout` and `transr` parameters can specify the same storage scheme: for example, the storage scheme for `matrix_layout = LAPACK_COL_MAJOR` and `transr = 'N'` is the same as that for `matrix_layout = LAPACK_ROW_MAJOR` and `transr = 'T'`.

Element $a_{ij}$ is stored as array element $ap[l]$ where the mapping of $(i,j)$ is defined in the following tables.

- **Column major layout:** `matrix_layout = LAPACK_COL_MAJOR`

<table>
<thead>
<tr>
<th>transr</th>
<th>uplo</th>
<th>N</th>
<th>$l(i, j) =$</th>
<th>$i$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>'U'</td>
<td>2*k</td>
<td>$(j-k)*(N+1) + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$i*(N+1) + j + k + 1$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2*k + 1</td>
<td>$(j-k)*N + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$i*N + j + k + 1$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'T' or 'C'</td>
<td>'U'</td>
<td>2*k</td>
<td>$j*(N+1) + i + 1$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(i-k)*(N+1) + j - k$</td>
<td>$k \leq i &lt; N$</td>
<td>$k \leq j \leq i$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2*k + 1</td>
<td>$j*N + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(i-k)*N + j - k - 1$</td>
<td>$k + 1 \leq i &lt; N$</td>
<td>$k + 1 \leq j \leq i$</td>
</tr>
<tr>
<td>'T' or 'C'</td>
<td>'U'</td>
<td>2*k</td>
<td>$i*k + j - k$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j + k + 1)*k + i$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2*k + 1</td>
<td>$i*(k+1) + j - k$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j + k + 1)*(k+1) + i$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'T' or 'C'</td>
<td>'U'</td>
<td>2*k</td>
<td>$(i+1)*k + j$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j - k)*k + i - k$</td>
<td>$k \leq i &lt; N$</td>
<td>$k \leq j \leq i$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2*k + 1</td>
<td>$(i+1)*k + j$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j - k - 1)*(k+1) + i - k$</td>
<td>$k + 1 \leq i &lt; N$</td>
<td>$k + 1 \leq j \leq i$</td>
</tr>
</tbody>
</table>

- **Row major layout:** `matrix_layout = LAPACK_ROW_MAJOR`
### Mathematical Notation for LAPACK Routines

Descriptions of LAPACK routines use the following notation:

<table>
<thead>
<tr>
<th>trans</th>
<th>uplo</th>
<th>$N$</th>
<th>$l(i, j) =$</th>
<th>$i$</th>
<th>$j$</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>'U'</td>
<td>2$k$</td>
<td>$i*k + j - k$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(k + j + 1)*k + i$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2$k$ + 1</td>
<td>$i*(k + 1) + j - k$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(k + j + 1)*(k + 1) + i$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2$k$</td>
<td>$(i + 1)*k + j$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j - k)*k + i - k$</td>
<td>$k \leq i &lt; N$</td>
<td>$k \leq j \leq i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2$k$ + 1</td>
<td>$i*(k + 1) + j$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(j - k - 1)*(k + 1) + i - k$</td>
<td>$k + 1 \leq i &lt; N$</td>
<td>$k + 1 \leq j \leq i$</td>
</tr>
<tr>
<td>'T' or 'C'</td>
<td>'U'</td>
<td>2$k$</td>
<td>$(j - k)*(N + 1) + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$i*(N + 1) + k + j + 1$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2$k$ + 1</td>
<td>$(j - k)*N + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$\max(i, k) \leq j &lt; N$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$i*N + k + j + 1$</td>
<td>$0 \leq i &lt; k$</td>
<td>$i \leq j &lt; k$</td>
</tr>
<tr>
<td>'L'</td>
<td></td>
<td>2$k$</td>
<td>$j*(N + 1) + i + 1$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(i - k)*(N + 1) + j - k$</td>
<td>$k \leq i &lt; N$</td>
<td>$k \leq j \leq i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2$k$ + 1</td>
<td>$j*N + i$</td>
<td>$0 \leq i &lt; N$</td>
<td>$0 \leq j \leq \min(i, k)$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$(i - k)*N + j - k - 1$</td>
<td>$k + 1 \leq i &lt; N$</td>
<td>$k + 1 \leq j \leq i$</td>
</tr>
</tbody>
</table>

**NOTE**

Although LAPACK accepts parameter values of zero for matrix size, in general the size of the array should be greater than or equal to max(1, $N^*(N + 1)/2$).
For an \( M \)-by-\( N \) matrix \( A \), denotes the conjugate transposed \( N \)-by-\( M \) matrix with elements:

\[
A_H^{H} = \overline{A_{ij}}
\]

For a real-valued matrix, \( A^H = A^T \).

The dot product of two vectors, defined as:

\[
x \cdot y = \sum_{i} x_i y_i
\]

Ax = b

A system of linear equations with an \( n \)-by-\( n \) matrix \( A = \{a_{ij}\} \), a right-hand side vector \( b = \{b_j\} \), and an unknown vector \( x = \{x_i\} \).

AX = B

A set of systems with a common matrix \( A \) and multiple right-hand sides. The columns of \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

|\( x |\)

the vector with elements |\( x_i |\) (absolute values of \( x_i \)).

|\( A |\)

the matrix with elements |\( a_{ij} |\) (absolute values of \( a_{ij} \)).

|\( |\sum_{i} x_i |\)\( \infty \ginfty\) = max |\( x_i |\)

The infinity-norm of the vector \( x \).

|\( |\sum_{i} a_{ij} |\)\( \infty \ginfty\) = max |\( a_{ij} |\)

The infinity-norm of the matrix \( A \).

|\( |\sum_{i} |\sum_{j} a_{ij} |\)\( \infty \ginfty\) = max |\( a_{ij} |\)

The one-norm of the matrix \( A \).

|\( |\sum_{i} x_i |\)\( 2 \ginfty\) = (\( \sum_{i} |x_i |^2 \))^{1/2}

The 2-norm of the vector \( x \): |\( x |\)\( 2 \ginfty\) = |\( x \) (see the definition for Euclidean norm in this section).

|\( |\sum_{i} x_i |\)\( 2 \ginfty\) = (\( \sum_{i} |x_i |^2 \))^{1/2}

The 2-norm (or spectral norm) of the matrix \( A \).

\[
|A|_2 = \max_{\|\mathbf{x}\| = 1} \|A\mathbf{x}\|_2
\]

The Euclidean norm of the matrix \( A \): \( |A|_E = (\sum_{i} |a_{ij} |^2)^{1/2} \).

\( \kappa (A) = |\|A\| \cdot |\|A^{-1}\|| \)

The condition number of the matrix \( A \).

\( \lambda_i \)

Eigenvalues of the matrix \( A \) (for the definition of eigenvalues, see Eigenvalue Problems).

\( \sigma_i \)

Singular values of the matrix \( A \). They are equal to square roots of the eigenvalues of \( A^T A \). (For more information, see Singular Value Decomposition).

---

**Error Analysis**

In practice, most computations are performed with rounding errors. Besides, you often need to solve a system \( Ax = b \) where the data (the elements of \( A \) and \( b \)) are not known exactly. Therefore, it is important to understand how the data errors and rounding errors can affect the solution \( x \).

**Data perturbations.** If \( x \) is the exact solution of \( Ax = b \), and \( x + \delta x \) is the exact solution of a perturbed problem \( (A + \delta A)(x + \delta x) = (b + \delta b) \), then this estimate, given up to linear terms of perturbations, holds:
where $A + \delta A$ is nonsingular and

$$\frac{\|\delta x\|}{\|x\|} \leq \kappa(A) \left( \frac{\|\delta A\|}{\|A\|} + \frac{\|\delta b\|}{\|b\|} \right)$$

In other words, relative errors in $A$ or $b$ may be amplified in the solution vector $x$ by a factor $\kappa(A) = \|A\|\|A^{-1}\|$ called the condition number of $A$.

Rounding errors have the same effect as relative perturbations $c(n)\epsilon$ in the original data. Here $\epsilon$ is the machine precision, defined as the smallest positive number $x$ such that $1 + x > 1$; and $c(n)$ is a modest function of the matrix order $n$. The corresponding solution error is

$$\|\delta x\|/\|x\| \leq c(n)\kappa(A)\epsilon.$$ (The value of $c(n)$ is seldom greater than $10n$.)

**NOTE**

Machine precision depends on the data type used. For example, it is usually defined in the `float.h` file as `FLT_EPSILON` the float datatype and `DBL_EPSILON` for the double datatype.

Thus, if your matrix $A$ is ill-conditioned (that is, its condition number $\kappa(A)$ is very large), then the error in the solution $x$ can also be large; you might even encounter a complete loss of precision. LAPACK provides routines that allow you to estimate $\kappa(A)$ (see Routines for Estimating the Condition Number) and also give you a more precise estimate for the actual solution error (see Refining the Solution and Estimating Its Error).

**LAPACK Linear Equation Routines**

This section describes routines for performing the following computations:
- factoring the matrix (except for triangular matrices)
- equilibrating the matrix (except for RFP matrices)
- solving a system of linear equations
- estimating the condition number of a matrix (except for RFP matrices)
- refining the solution of linear equations and computing its error bounds (except for RFP matrices)
- inverting the matrix.

To solve a particular problem, you can call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. For example, to solve a system of linear equations with a general matrix, call `?getrf` (LU factorization) and then `?getrs` (computing the solution). Then, call `?gerfs` to refine the solution and get the error bounds. Alternatively, use the driver routine `?gesvx` that performs all these tasks in one call.

**LAPACK Linear Equation Computational Routines**

Table "Computational Routines for Systems of Equations with Real Matrices" lists the LAPACK computational routines for factorizing, equilibrating, and inverting real matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error. Table "Computational Routines for Systems of Equations with Complex Matrices" lists similar routines for complex matrices.

**Computational Routines for Systems of Equations with Real Matrices**

<table>
<thead>
<tr>
<th>Matrix type, storage scheme</th>
<th>Factorize matrix</th>
<th>Equilibrate matrix</th>
<th>Solve system</th>
<th>Condition number</th>
<th>Estimate error</th>
<th>Invert matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix type, storage scheme</td>
<td>Factorize matrix</td>
<td>Equilibrate matrix</td>
<td>Solve system</td>
<td>Condition number</td>
<td>Estimate error</td>
<td>Invert matrix</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------</td>
<td>--------------------</td>
<td>--------------</td>
<td>------------------</td>
<td>---------------</td>
<td>---------------</td>
</tr>
<tr>
<td>general tridiagonal</td>
<td>?gttrf</td>
<td></td>
<td>?gttrs</td>
<td>?gtcon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diagonally dominant tridiagonal</td>
<td>?dttrfb</td>
<td></td>
<td>?dttrsb</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symmetric positive-definite, RFP storage</td>
<td>?pftrf</td>
<td></td>
<td>?pftrs</td>
<td></td>
<td>?pftri</td>
<td></td>
</tr>
<tr>
<td>symmetric positive-definite, tridiagonal</td>
<td>?pttrf</td>
<td></td>
<td>?pttrs</td>
<td>?ptcon</td>
<td>?pttri</td>
<td></td>
</tr>
<tr>
<td></td>
<td>?sytrf_aa</td>
<td>?sytrf3</td>
<td>?sytrf_a</td>
<td>?sytri2x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
mkl_?spftr2, mkl_?spftrtx
| triangular, packed storage  | ?tptrs          |                    | ?tpcon       |                  |               |               |
| triangular, RFP storage     |                  |                    |               |                  |               | ?tftri        |
| triangular band             | ?tbtrs          |                    | ?tbcon       |                  |               | ?tbrfs        |

**Computational Routines for Systems of Equations with Complex Matrices**

<table>
<thead>
<tr>
<th>Matrix type, storage scheme</th>
<th>Factorize matrix</th>
<th>Equilibrate matrix</th>
<th>Solve system</th>
<th>Condition number</th>
<th>Estimate error</th>
<th>Invert matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>?geequeb</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix type, storage scheme</td>
<td>Factorize matrix</td>
<td>Equilibrate matrix</td>
<td>Solve system</td>
<td>Condition number</td>
<td>Estimate error</td>
<td>Invert matrix</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>------------------</td>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hermitian positive-definite, RFP storage</td>
<td>?pftrf</td>
<td></td>
<td>?pftrs</td>
<td></td>
<td></td>
<td>?pftri</td>
</tr>
<tr>
<td></td>
<td>?hetrf_aa</td>
<td></td>
<td>?hets_3</td>
<td></td>
<td></td>
<td>?hetri2x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>?hets_aa</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>?sytrs3</td>
<td></td>
<td></td>
<td>?sytri2x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mkl_?spffrt2, mkl_?spffrtx</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular</td>
<td>?trtrs</td>
<td></td>
<td>?trcon</td>
<td></td>
<td>?ttrfs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular, packed storage</td>
<td>?tptrs</td>
<td></td>
<td>?tpcon</td>
<td></td>
<td>?tprf</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular, RFP storage</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>?tbtrf</td>
<td></td>
<td>?tbcon</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular band</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LAPACK Routines
Matrix Factorization: LAPACK Computational Routines

This section describes the LAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization
- Cholesky factorization of real symmetric positive-definite matrices
- Cholesky factorization of real symmetric positive-definite matrices with pivoting
- Cholesky factorization of Hermitian positive-definite matrices
- Cholesky factorization of Hermitian positive-definite matrices with pivoting
- Bunch-Kaufman factorization of real and complex symmetric matrices
- Bunch-Kaufman factorization of Hermitian matrices.

You can compute:

- the LU factorization using full and band storage of matrices
- the Cholesky factorization using full, packed, RFP, and band storage
- the Bunch-Kaufman factorization using full and packed storage.

`?getrf`

*Computes the LU factorization of a general m-by-n matrix.*

**Syntax**

```c
lapack_int LAPACKE_sgetrf (int matrix_layout, lapack_int m, lapack_int n, float * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_dgetrf (int matrix_layout, lapack_int m, lapack_int n, double * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_cgetrf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_zgetrf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**

The routine computes the LU factorization of a general m-by-n matrix $A$ as

$$A = P*L*U,$$

where $P$ is a permutation matrix, $L$ is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and $U$ is upper triangular (upper trapezoidal if $m < n$). The routine uses partial pivoting, with row interchanges.

**NOTE**

This routine supports the Progress Routine feature. See Progress Functions section for details.

**Input Parameters**

- `matrix_layout`: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
m
n
a
lda

The number of rows in the matrix A (m≥ 0).
The number of columns in A; n≥ 0.
Array, size at least \( \max(1, lda\cdot n) \) for column-major layout or \( \max(1, lda\cdot m) \) for row-major layout. Contains the matrix A.
The leading dimension of array a, which must be at least \( \max(1, m) \) for column-major layout or \( \max(1, n) \) for row-major layout.

Output Parameters

a
Overwritten by L and U. The unit diagonal elements of L are not stored.
ipiv
Array, size at least \( \max(1, \min(m, n)) \). Contains the pivot indices; for \( 1 \leq i \leq \min(m, n) \), row i was interchanged with row ipiv(i).

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, \( u_{ii} \) is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The computed L and U are the exact factors of a perturbed matrix A + E, where

\[
|E| \leq c(\min(m,n))\epsilon \|L\|\|U\|
\]

\( c(n) \) is a modest linear function of n, and \( \epsilon \) is the machine precision.

The approximate number of floating-point operations for real flavors is

\[
\begin{align*}
(2/3)n^3 & \quad \text{If } m = n, \\
(1/3)n^2(3m-n) & \quad \text{If } m>n, \\
(1/3)m^2(3n-m) & \quad \text{If } m<n.
\end{align*}
\]

The number of operations for complex flavors is four times greater.

After calling this routine with \( m = n \), you can call the following:

?-getrs to solve \( A^r X = B \) or \( A^t X = B \) or \( A^H X = B \)

?-gecon to estimate the condition number of A

?-getri to compute the inverse of A.

See Also

mkl_progress

Matrix Storage Schemes
mkl_getrfnp
Computes the LU factorization of a general m-by-n matrix without pivoting.

Syntax

```c
lapack_int LAPACKE_mkl_sgetrfnp (int matrix_layout, lapack_int m, lapack_int n,
                         float * a, lapack_int lda);

lapack_int LAPACKE_mkl_dgetrfnp (int matrix_layout, lapack_int m, lapack_int n,
                         double * a, lapack_int lda);

lapack_int LAPACKE_mkl_cgetrfnp (int matrix_layout, lapack_int m, lapack_int n,
                         lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_mkl_zgetrfnp (int matrix_layout, lapack_int m, lapack_int n,
                         lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine computes the LU factorization of a general m-by-n matrix A as

\[ A = L*U, \]

where L is lower triangular with unit-diagonal elements (lower trapezoidal if \( m > n \)) and U is upper triangular (upper trapezoidal if \( m < n \)). The routine does not use pivoting.

Input Parameters

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `m`
  The number of rows in the matrix A \((m \geq 0)\).

- `n`
  The number of columns in A; \( n \geq 0 \).

- `a`
  Array, size at least max(1, \( lda*n \)) for column-major layout or max(1, \( lda*m \)) for row-major layout. Contains the matrix A.

- `lda`
  The leading dimension of array a, which must be at least max(1, \( m \)) for column-major layout or max(1, \( n \)) for row-major layout.

Output Parameters

- `a`
  Overwritten by L and U. The unit diagonal elements of L are not stored.

Return Values

This function returns a value `info`.

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), \( u_{ii} \) is 0. The factorization has been completed, but U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.
Application Notes

The approximate number of floating-point operations for real flavors is

\[(2/3)n^3\]  \quad \text{If } m = n,

\[(1/3)n^2(3m-n)\]  \quad \text{If } m > n,

\[(1/3)m^2(3n-m)\]  \quad \text{If } m < n.

The number of operations for complex flavors is four times greater.

After calling this routine with \(m = n\), you can call the following:

mkl\_?getrinp to compute the inverse of \(A\)

See Also

mkl\_progress

Matrix Storage Schemes

mkl\_?getrfnpi

Performs LU factorization (complete or incomplete) of a general matrix without pivoting.

Syntax

\[
\begin{align*}
\text{lapack_int LAPACKE_mkl_sgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,}\ \\
\text{lapack_int nfact, float* a, lapack_int lda);}
\text{lapack_int LAPACKE_mkl_dgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,}\ \\
\text{lapack_int nfact, double* a, lapack_int lda);}
\text{lapack_int LAPACKE_mkl_cgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,}\ \\
\text{lapack_int nfact, lapack_complex_float* a, lapack_int lda);}
\text{lapack_int LAPACKE_mkl_zgetrfnpi (int matrix_layout, lapack_int m, lapack_int n,}\ \\
\text{lapack_int nfact, lapack_complex_double* a, lapack_int lda);}\
\end{align*}
\]

Include Files

- mkl.h

Description

The routine computes the LU factorization of a general \(m\)-by-\(n\) matrix \(A\) without using pivoting. It supports incomplete factorization. The factorization has the form:

\[A = L^*U,\]

where \(L\) is lower triangular with unit diagonal elements (lower trapezoidal if \(m > n\)) and \(U\) is upper triangular (upper trapezoidal if \(m < n\)).

Incomplete factorization has the form:

\[A = L \ast U + \hat{A}\]

where \(L\) is lower trapezoidal with unit diagonal elements, \(U\) is upper trapezoidal, and \(\hat{A}\) is the unfactored part of matrix \(A\). See the application notes section for further details.
NOTE
Use \texttt{?getrf} if it is possible that the matrix is not diagonal dominant.

**Input Parameters**

\begin{itemize}
  \item \texttt{matrix\_layout} 
    Specifies whether matrix storage layout is row major (\texttt{LAPACK\_ROW\_MAJOR}) or column major (\texttt{LAPACK\_COL\_MAJOR}).
  \item \texttt{m} 
    The number of rows in matrix \(A\); \(m \geq 0\).
  \item \texttt{n} 
    The number of columns in matrix \(A\); \(n \geq 0\).
  \item \texttt{nfact} 
    The number of rows and columns to factor; \(0 \leq nfact \leq \min(m, n)\). Note that if \(nfact < \min(m, n)\), incomplete factorization is performed.
  \item \texttt{a} 
    Array of size at least \(lda\ast n\) for column major layout and at least \(lda\ast m\) for row major layout. Contains the matrix \(A\).
  \item \texttt{lda} 
    The leading dimension of array \(a\). \(lda \geq \max(1, m)\) for column major layout and \(lda \geq \max(1, n)\) for row major layout.
\end{itemize}

**Output Parameters**

\texttt{a} 
Overwritten by \(L\) and \(U\). The unit diagonal elements of \(L\) are not stored.

When incomplete factorization is specified by setting \(nfact < \min(m, n)\), a also contains the unfactored submatrix \(\tilde{A}_{22}\). See the application notes section for further details.

**Return Values**

This function returns a value \(info\).

If \(info=0\), the execution is successful.

If \(info = -i\), the \(i\)-th parameter had an illegal value.

If \(info = i\), \(u_i\) is 0. The requested factorization has been completed, but \(U\) is exactly singular. Division by 0 will occur if factorization is completed and factor \(U\) is used for solving a system of linear equations.

**Application Notes**

The computed \(L\) and \(U\) are the exact factors of a perturbed matrix \(A + E\), with

\[ |E| \leq c(n) \varepsilon |L||U| \]

where \(c(n)\) is a modest linear function of \(n\), and \(\varepsilon\) is the machine precision.

The approximate number of floating-point operations for real flavors is

\begin{align*}
(2/3)n^3 & \quad \text{if} \quad m = n = nfact \\
(1/3)n^2(3m-n) & \quad \text{if} \quad m > n = nfact \\
(1/3)m^2(3n-m) & \quad \text{if} \quad m = nfact < n \\
(2/3)n^3 - (n-nfact)^3 & \quad \text{if} \quad m = n, nfact < \min(m, n)
\end{align*}
The number of operations for complex flavors is four times greater.

When incomplete factorization is specified, the first \( n_{\text{fact}} \) rows and columns are factored, with the update of the remaining rows and columns of \( A \) as follows:

If matrix \( A \) is represented as a block 2-by-2 matrix:

\[
A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}
\]

where

- \( A_{11} \) is a square matrix of order \( n_{\text{fact}} \),
- \( A_{21} \) is an \((m - n_{\text{fact}})\)-by-\( n_{\text{fact}} \) matrix,
- \( A_{12} \) is an \( n_{\text{fact}} \)-by-\((n - n_{\text{fact}})\) matrix, and
- \( A_{22} \) is an \((m - n_{\text{fact}})\)-by-\((n - n_{\text{fact}})\) matrix.

The result is

\[
A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix} \begin{bmatrix} U_1 & U_2 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}
\]

\( L_1 \) is a lower triangular square matrix of order \( n_{\text{fact}} \) with unit diagonal and \( U_1 \) is an upper triangular square matrix of order \( n_{\text{fact}} \). \( L_1 \) and \( U_1 \) result from LU factorization of matrix \( A_{11} \): \( A_{11} = L_1 U_1 \).

\( L_2 \) is an \((m - n_{\text{fact}})\)-by-\( n_{\text{fact}} \) matrix and \( L_2 = A_{21} U_1^{-1} \). \( U_2 \) is an \( n_{\text{fact}} \)-by-\((n - n_{\text{fact}})\) matrix and \( U_2 = L_1^{-1} A_{12} \).

\( A_{22} \) is an \((m - n_{\text{fact}})\)-by-\((n - n_{\text{fact}})\) matrix and \( \tilde{A}_{22} = A_{22} - L_2 U_2 \).
On exit, elements of the upper triangle \( U_1 \) are stored in place of the upper triangle of block \( A_{11} \) in array \( a \); elements of the lower triangle \( L_1 \) are stored in the lower triangle of block \( A_{11} \) in array \( a \) (unit diagonal elements are not stored). Elements of \( L_2 \) replace elements of \( A_{21} \); \( U_2 \) replaces elements of \( A_{12} \) and \( \tilde{A}_{22} \) replaces elements of \( A_{22} \).

\[ \begin{array}{c}
\text{input} \quad \begin{array}{ccc}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{array} \\
\downarrow \quad m \\
\downarrow \quad n \\
\end{array} \quad \begin{array}{ccc}
\text{output} \\
U_1 & \ \ \ \ \ \ \ \ \ \ U_2 \\
L_1 & \ \ \ \ \ \ \ \ \ \ \ L_2 \\
& \tilde{A}_{22}
\end{array} \]\n
`getrf2`

*Computes LU factorization using partial pivoting with row interchanges.*

**Syntax**

```c
lapack_int LAPACKE_sgetrf2 (int matrix_layout, lapack_int m, lapack_int n, float * a, lapack_int lda, lapack_int * ipiv);
```

```c
lapack_int LAPACKE_dgetrf2 (int matrix_layout, lapack_int m, lapack_int n, double * a, lapack_int lda, lapack_int * ipiv);
```

```c
lapack_int LAPACKE_cgetrf2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);
```

```c
lapack_int LAPACKE_zgetrf2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**

`getrf2` computes an LU factorization of a general \( m \)-by-\( n \) matrix \( A \) using partial pivoting with row interchanges.

The factorization has the form

\[
A = P \cdot L \cdot U
\]

where \( P \) is a permutation matrix, \( L \) is lower triangular with unit diagonal elements (lower trapezoidal if \( m > n \)), and \( U \) is upper triangular (upper trapezoidal if \( m < n \)).

This is the recursive version of the algorithm. It divides the matrix into four submatrices:

\[
A = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix}
\]
where $A_{11}$ is $n_1$ by $n_1$ and $A_{22}$ is $n_2$ by $n_2$ with $n_1 = \min(m, n)$, and $n_2 = n - n_1$.

The subroutine calls itself to factor $\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix}$, do the swaps on $\begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix}$, solve $A_{12}$, update $A_{22}$, then it calls itself to factor $A_{22}$ and do the swaps on $A_{21}$.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **$m$**
  The number of rows of the matrix A. $m \geq 0$.

- **$n$**
  The number of columns of the matrix A. $n \geq 0$.

- **$a$**
  Array, size $lda \times n$.
  On entry, the $m$-by-$n$ matrix to be factored.

- **$lda$**
  The leading dimension of the array $a$. $lda \geq \max(1, m)$.

**Output Parameters**

- **$a$**
  On exit, the factors $L$ and $U$ from the factorization $A = P \times L \times U$; the unit diagonal elements of $L$ are not stored.

- **$ipiv$**
  Array, size $(\min(m, n))$.
  The pivot indices; for $1 \leq i \leq \min(m, n)$, row $i$ of the matrix was interchanged with row $ipiv[i - 1]$.

**Return Values**

This function returns a value $info$.

- $= 0$: successful exit
- $< 0$: if $info = -i$, the $i$-th argument had an illegal value.
- $> 0$: if $info = i$, $U_{ii}$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, and division by zero will occur if it is used to solve a system of equations.

`gbtrf`

*Computes the LU factorization of a general $m$-by-$n$ band matrix.*

**Syntax**

```c
lapack_int LAPACKE_sgbtrf (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, float * ab, lapack_int ldab, lapack_int * ipiv);
lapack_int LAPACKE_dgbtrf (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, double * ab, lapack_int ldab, lapack_int * ipiv);
lapack_int LAPACKE_cgbtrf (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, lapack_complex_float * ab, lapack_int ldab, lapack_int * ipiv);
lapack_int LAPACKE_zgbtrf (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, lapack_complex_double * ab, lapack_int ldab, lapack_int * ipiv);
```
Include Files

- mkl.h

Description

The routine forms the $LU$ factorization of a general $m$-by-$n$ band matrix $A$ with $kl$ non-zero subdiagonals and $ku$ non-zero superdiagonals, that is,

$$ A = P*L*U, $$

where $P$ is a permutation matrix; $L$ is lower triangular with unit diagonal elements and at most $kl$ non-zero elements in each column; $U$ is an upper triangular band matrix with $kl + ku$ superdiagonals. The routine uses partial pivoting, with row interchanges (which creates the additional $kl$ superdiagonals in $U$).

**NOTE**

This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **m**
  The number of rows in matrix $A$; $m \geq 0$.

- **n**
  The number of columns in matrix $A$; $n \geq 0$.

- **kl**
  The number of subdiagonals within the band of $A$; $kl \geq 0$.

- **ku**
  The number of superdiagonals within the band of $A$; $ku \geq 0$.

- **ab**
  Array, size at least max(1, $ldab*n$) for column-major layout or max(1, $ldab*m$) for row-major layout. The array $ab$ contains the matrix $A$ in band storage as described in Band Storage.

- **ldab**
  The leading dimension of the array $ab$. ($ldab \geq 2*kl + ku + 1$)

Output Parameters

- **ab**
  Overwritten with elements of $L$ and $U$. $U$ is stored as an upper triangular band matrix with $kl + ku$ superdiagonals, and $L$ is stored as a lower triangular band matrix with $kl$ subdiagonals (diagonal unit values are not stored). Since the output array has more nonzero elements than the initial matrix $A$, there are limitations on the value of $ldab$ and the placement of elements of $A$ in array $ab$.

  See Application Notes below for further details.

- **ipiv**
  Array, size at least max(1, min($m$, $n$)). The pivot indices; for $1 \leq i \leq \min(m, n)$, row $i$ was interchanged with row $ipiv(i)$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.
If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, $u_{ii}$ is 0. The factorization has been completed, but $U$ is exactly singular. Division by 0 will occur if you use the factor $U$ for solving a system of linear equations.

**Application Notes**

The computed $L$ and $U$ are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(kl+ku+1)\epsilon P \|L\|\|U\|$$

$c(k)$ is a modest linear function of $k$, and $\epsilon$ is the machine precision.

The total number of floating-point operations for real flavors varies between approximately $2n(ku+1)kl$ and $2n(kl+ku+1)kl$. The number of operations for complex flavors is four times greater. All these estimates assume that $kl$ and $ku$ are much less than $\min(m,n)$.

As described in Band Storage, storage of a band matrix can be considered in two steps: packing band matrix elements into a matrix $AB$, then storing the elements in a linear array $ab$ using a full storage scheme. The effect of the ?gbtrf routine on matrix $AB$ is illustrated by this example, for $m = n = 6, kl = 2, ku = 1$.

- **matrix_layout = LAPACK_COL_MAJOR**

On entry:

$$AB = \begin{bmatrix}
* & * & * & + & + & + \\
* & * & + & + & + & + \\
* & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\
a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{65} \\
a_{21} & a_{32} & a_{43} & a_{54} & a_{64} & * \\
a_{31} & a_{42} & a_{53} & a_{64} & * & * \\
\end{bmatrix}$$

On exit:

$$AB = \begin{bmatrix}
* & * & * & u_{14} & u_{25} & u_{36} \\
* & u_{13} & u_{24} & u_{35} & u_{45} & u_{56} \\
* & u_{12} & u_{23} & u_{34} & u_{44} & u_{55} & u_{65} \\
* & u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\
* & l_{12} & l_{23} & l_{34} & l_{44} & l_{54} & l_{66} \\
* & l_{11} & l_{22} & l_{33} & l_{43} & l_{53} & l_{64} \\
\end{bmatrix}$$

- **matrix_layout = LAPACK_ROW_MAJOR**

On entry:

$$AB = \begin{bmatrix}
* & * & a_{11} & a_{12} & + & + \\
* & a_{21} & a_{22} & a_{23} & + & + \\
a_{31} & a_{32} & a_{33} & a_{34} & + & + \\
a_{42} & a_{43} & a_{44} & a_{45} & + & * \\
a_{53} & a_{54} & a_{55} & a_{56} & * & * \\
a_{64} & a_{65} & a_{66} & * & * & * \\
\end{bmatrix}$$

On exit:

$$AB = \begin{bmatrix}
* & * & u_{11} & u_{12} & u_{13} & u_{14} \\
* & l_{21} & u_{22} & u_{23} & u_{24} & u_{25} \\
* & l_{31} & l_{32} & u_{33} & u_{34} & u_{35} & u_{36} \\
* & l_{42} & l_{43} & u_{44} & u_{45} & u_{46} & * \\
* & l_{53} & l_{54} & u_{55} & u_{56} & * & * \\
* & l_{64} & l_{65} & u_{66} & * & * & * \\
\end{bmatrix}$$

Elements marked * are not used; elements marked + need not be set on entry, but are required by the routine to store elements of $U$ because of fill-in resulting from the row interchanges.

After calling this routine with $m = n$, you can call the following routines:

- **gbtrs** to solve $A*X = B$ or $A^T*X = B$ or $A^H*X = B$
- **gbcon** to estimate the condition number of $A$.

**See Also**

mkl_progress
Matrix Storage Schemes

\texttt{?gttrf}

\textit{Computes the LU factorization of a tridiagonal matrix.}

\textbf{Syntax}

\begin{verbatim}
lapack_int LAPACKE_sgttrf (lapack_int n, float * dl, float * d, float * du, float * du2, lapack_int * ipiv);

lapack_int LAPACKE_dgttrf (lapack_int n, double * dl, double * d, double * du, double * du2, lapack_int * ipiv);

lapack_int LAPACKE_cgttrf (lapack_int n, lapack_complex_float * dl, lapack_complex_float * d, lapack_complex_float * du, lapack_complex_float * du2, lapack_int * ipiv);

lapack_int LAPACKE_zgttrf (lapack_int n, lapack_complex_double * dl, lapack_complex_double * d, lapack_complex_double * du, lapack_complex_double * du2, lapack_int * ipiv);
\end{verbatim}

\textbf{Include Files}

- mkl.h

\textbf{Description}

The routine computes the \textit{LU} factorization of a real or complex tridiagonal matrix $A$ using elimination with partial pivoting and row interchanges.

The factorization has the form

$$A = L \cdot U,$$

where $L$ is a product of permutation and unit lower bidiagonal matrices and $U$ is upper triangular with nonzeros in only the main diagonal and first two superdiagonals.

\textbf{Input Parameters}

- \textit{n}
  - The order of the matrix $A$; \(n \geq 0\).

- \textit{dl, d, du}
  - Arrays containing elements of $A$.
  - The array \textit{dl} of dimension \((n - 1)\) contains the subdiagonal elements of $A$.
  - The array \textit{d} of dimension \(n\) contains the diagonal elements of $A$.
  - The array \textit{du} of dimension \((n - 1)\) contains the superdiagonal elements of $A$.

\textbf{Output Parameters}

- \textit{dl}
  - Overwritten by the \((n-1)\) multipliers that define the matrix $L$ from the \textit{LU} factorization of $A$.

- \textit{d}
  - Overwritten by the \(n\) diagonal elements of the upper triangular matrix $U$ from the \textit{LU} factorization of $A$.

- \textit{du}
  - Overwritten by the \((n-1)\) elements of the first superdiagonal of $U$. 
du2

Array, dimension \((n-2)\). On exit, \(du2\) contains \((n-2)\) elements of the second superdiagonal of \(U\).

ipiv

Array, dimension \((n)\). The pivot indices: for \(1 \leq i \leq n\), row \(i\) was interchanged with row \(ipiv[i-1]\). \(ipiv[i-1]\) is always \(i\) or \(i+1\); \(ipiv[i-1] = i\) indicates a row interchange was not required.

Return Values

This function returns a value \(info\).

If \(info = 0\), the execution is successful.

If \(info = -i\), parameter \(i\) had an illegal value.

If \(info = i\), \(u_{ii}\) is 0. The factorization has been completed, but \(U\) is exactly singular. Division by zero will occur if you use the factor \(U\) for solving a system of linear equations.

Application Notes

\(?\)gbtrs

to solve \(A^{}X = B\) or \(A^T\cdot X = B\) or \(A^H\cdot X = B\)

\(?\)gbcon

to estimate the condition number of \(A\).

\(?\)dttrfb

**Computes the factorization of a diagonally dominant tridiagonal matrix.**

Syntax

```c
void sdttrfb (const MKL_INT * n , float * dl , float * d , const float * du , MKL_INT * info );
void ddttrfb (const MKL_INT * n , double * dl , double * d , const double * du , MKL_INT * info );
void cdttrfb (const MKL_INT * n , MKL_Complex8 * dl , MKL_Complex8 * d , const MKL_Complex8 * du , MKL_INT * info );
void zdttrfb_ (const MKL_INT * n , MKL_Complex16 * dl , MKL_Complex16 * d , const MKL_Complex16 * du , MKL_INT * info );
```

Include Files

- mkl.h

Description

The \(?\)dttrfb routine computes the factorization of a real or complex tridiagonal matrix \(A\) with the BABE (Burning At Both Ends) algorithm without pivoting. The factorization has the form

\[
A = L_1 \cdot U \cdot L_2
\]

where

- \(L_1\) and \(L_2\) are unit lower bidiagonal with \(k\) and \(n - k - 1\) subdiagonal elements, respectively, where \(k = n/2\), and
- \(U\) is an upper bidiagonal matrix with nonzeros in only the main diagonal and first superdiagonal.
Input Parameters

\( n \)  
The order of the matrix \( A \); \( n \geq 0 \).

\( dl, d, du \)  
Arrays containing elements of \( A \).

The array \( dl \) of dimension \((n - 1)\) contains the subdiagonal elements of \( A \).

The array \( d \) of dimension \( n \) contains the diagonal elements of \( A \).

The array \( du \) of dimension \((n - 1)\) contains the superdiagonal elements of \( A \).

Output Parameters

\( dl \)  
Overwritten by the \((n - 1)\) multipliers that define the matrix \( L \) from the \( LU \) factorization of \( A \).

\( d \)  
Overwritten by the \( n \) diagonal element reciprocals of the upper triangular matrix \( U \) from the factorization of \( A \).

\( du \)  
Overwritten by the \((n-1)\) elements of the superdiagonal of \( U \).

\( info \)  
If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = i \), \( u_{ii} \) is 0. The factorization has been completed, but \( U \) is exactly singular. Division by zero will occur if you use the factor \( U \) for solving a system of linear equations.

Application Notes

A diagonally dominant tridiagonal system is defined such that \(|d_i| > |dl_{i-1}| + |du_i|\) for any \( i \):

\[
1 < i < n, \text{ and } |d_1| > |du_1|, |d_n| > |dl_{n-1}|
\]

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems are free from the numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see \(?gttrf\)). The diagonally dominant systems are much faster than the canonical systems.

**NOTE**

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the \(?dttrfb\) factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

\?potrf

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix.*

**Syntax**

```c
lapack_int LAPACKE_spotrf (int matrix_layout , char uplo , lapack_int n , float * a , lapack_int lda );
```
lapack_int LAPACKE_dpotrf (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda);
lapack_int LAPACKE_cpotrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda);
lapack_int LAPACKE_zpotrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda);

Include Files
• mkl.h

Description
The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix $A$:

- $A = U^T \cdot U$ for real data, $A = U^H \cdot U$ for complex data if $\text{uplo} = 'U'$
- $A = L \cdot L^T$ for real data, $A = L \cdot L^H$ for complex data if $\text{uplo} = 'L'$

where $L$ is a lower triangular matrix and $U$ is upper triangular.

NOTE
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters
matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
uplo Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
- If $\text{uplo} = 'U'$, the array $a$ stores the upper triangular part of the matrix $A$, and the strictly lower triangular part of the matrix is not referenced.
- If $\text{uplo} = 'L'$, the array $a$ stores the lower triangular part of the matrix $A$, and the strictly upper triangular part of the matrix is not referenced.
n Specifies the order of the matrix $A$. The value of $n$ must be at least zero.
a Array, size $\text{max}(1, \text{lda} \cdot n)$. The array $a$ contains either the upper or the lower triangular part of the matrix $A$ (see $\text{uplo}$).
lda The leading dimension of $a$. Must be at least $\text{max}(1, n)$.

Output Parameters
a The upper or lower triangular part of $a$ is overwritten by the Cholesky factor $U$ or $L$, as specified by $\text{uplo}$.

Return Values
This function returns a value $\text{info}$.
If info=0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix A.

**Application Notes**

If uplo = 'U', the computed factor U is the exact factor of a perturbed matrix A + E, where

\[
|E| \leq c(n)\varepsilon |U^H| |U|, |e_{ij}| \leq c(n)\varepsilon \sqrt{a_{ii}a_{jj}}
\]

\(c(n)\) is a modest linear function of \(n\), and \(\varepsilon\) is the machine precision.

A similar estimate holds for uplo = 'L'.

The total number of floating-point operations is approximately \((1/3)n^3\) for real flavors or \((4/3)n^3\) for complex flavors.

After calling this routine, you can call the following routines:

- ?potrs to solve \(A^*X = B\)
- ?pocon to estimate the condition number of \(A\)
- ?potri to compute the inverse of \(A\).

**See Also**

mkl_progress

**Matrix Storage Schemes**

??potrf2

Computes Cholesky factorization using a recursive algorithm.

**Syntax**

```c
lapack_int LAPACKE_spotrf2 (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda);
lapack_int LAPACKE_dpotrf2 (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda);
lapack_int LAPACKE_cpotrf2 (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda);
lapack_int LAPACKE_zpotrf2 (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda);
```

**Include Files**

- mkl.h

**Description**

??potrf2 computes the Cholesky factorization of a real or complex symmetric positive definite matrix A using the recursive algorithm.

The factorization has the form for real flavors:
\[ A = U^T U, \text{ if } \text{uplo} = 'U', \text{ or } \]
\[ A = L \ast L^T, \text{ if } \text{uplo} = 'L', \]
for complex flavors:
\[ A = U^H U, \text{ if } \text{uplo} = 'U', \]
\[ \text{or } A = L \ast L^H, \text{ if } \text{uplo} = 'L', \]
where \( U \) is an upper triangular matrix and \( L \) is lower triangular.

This is the recursive version of the algorithm. It divides the matrix into four submatrices:
\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}
\]
where \( A_{11} \) is \( n_1 \) by \( n_1 \) and \( A_{22} \) is \( n_2 \) by \( n_2 \), with \( n_1 = n/2 \) and \( n_2 = n-n_1 \).

The subroutine calls itself to factor \( A_{11} \). Update and scale \( A_{21} \) or \( A_{12} \), update \( A_{22} \) then call itself to factor \( A_{22} \).

**Input Parameters**

\[ \text{matrix\_layout} \]
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\[ \text{uplo} \]
= 'U': Upper triangle of \( A \) is stored;
= 'L': Lower triangle of \( A \) is stored.

\[ n \]
The order of the matrix \( A \).
\( n \geq 0 \).

\[ a \]
Array, size (\( \text{lda} \ast n \)).
On entry, the symmetric matrix \( A \).
If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( a \) contains the upper triangular part of the matrix \( A \), and the strictly lower triangular part of \( a \) is not referenced.
If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( a \) contains the lower triangular part of the matrix \( A \), and the strictly upper triangular part of \( a \) is not referenced.

\[ \text{lda} \]
The leading dimension of the array \( a \).
\( \text{lda} \geq \max(1,n) \).

**Output Parameters**

\[ a \]
On exit, if \( \text{info} = 0 \), the factor \( U \) or \( L \) from the Cholesky factorization.
For real flavors:
\[ A = U^T U \text{ or } A = L \ast L^T; \]
For complex flavors:
\[ A = U^H U \text{ or } A = L \ast L^H. \]

**Return Values**
This function returns a value \( \text{info} \).
= 0: successful exit
< 0: if info = -i, the i-th argument had an illegal value
> 0: if info = i, the leading minor of order i is not positive definite, and the factorization could not be completed.

*pstrf
Computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix.

Syntax

```c
lapack_int LAPACKE_spstrf( int matrix_layout, char uplo, lapack_int n, float* a, lapack_int lda, lapack_int* piv, lapack_int* rank, float tol);

lapack_int LAPACKE_dpstrf( int matrix_layout, char uplo, lapack_int n, double* a, lapack_int lda, lapack_int* piv, lapack_int* rank, double tol);

lapack_int LAPACKE_cpstrf( int matrix_layout, char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* piv, lapack_int* rank, float tol);

lapack_int LAPACKE_zpstrf( int matrix_layout, char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* piv, lapack_int* rank, double tol);
```

Include Files

- mkl.h

Description

The routine computes the Cholesky factorization with complete pivoting of a real symmetric (complex Hermitian) positive semidefinite matrix. The form of the factorization is:

- \( P^T \cdot A \cdot P = U^T \cdot U \) for real flavors, where \( P \) is a permutation matrix stored as vector \( \text{piv} \), and \( U \) is an upper triangular matrix.

- \( P^T \cdot A \cdot P = U^H \cdot U \) for complex flavors.

- \( P^T \cdot A \cdot P = L \cdot L^T \) for real flavors.

- \( P^T \cdot A \cdot P = L \cdot L^H \) for complex flavors.

where \( P \) is a permutation matrix stored as vector \( \text{piv} \), and \( U \) and \( L \) are upper and lower triangular matrices, respectively.

This algorithm does not attempt to check that \( A \) is positive semidefinite. This version of the algorithm calls level 3 BLAS.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of \( A \) is stored:
  If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular part of the matrix \( A \), and the strictly lower triangular part of the matrix is not referenced.
If uplo = 'L', the array \( a \) stores the lower triangular part of the matrix \( A \), and the strictly upper triangular part of the matrix is not referenced.

\( n \)

The order of matrix \( A \); \( n \geq 0 \).

\( a \)

Array \( a \), size \( \max(1,lda\ast n) \). The array \( a \) contains either the upper or the lower triangular part of the matrix \( A \) (see uplo).

\( tol \)

User defined tolerance. If \( tol < 0 \), then \( n\ast\varepsilon\ast\max(A_{k,k}) \), where \( \varepsilon \) is the machine precision, will be used (see Error Analysis for the definition of machine precision). The algorithm terminates at the \((k-1)\)-st step, if the pivot \( \leq tol \).

\( lda \)

The leading dimension of \( a \); at least \( \max(1, n) \).

**Output Parameters**

\( a \)

If info = 0, the factor \( U \) or \( L \) from the Cholesky factorization is as described in Description.

\( piv \)

Array, size at least \( \max(1, n) \). The array \( piv \) is such that the nonzero entries are \( P_{piv[k-1],k} \) \((1 \leq k \leq n)\).

\( rank \)

The rank of \( a \) given by the number of steps the algorithm completed.

**Return Values**

This function returns a value info.

If info = 0, the execution is successful.

If info = -k, the k-th argument had an illegal value.

If info > 0, the matrix \( A \) is either rank deficient with a computed rank as returned in rank, or is not positive semidefinite.

**See Also**

Matrix Storage Schemes

?pftrf

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using the Rectangular Full Packed (RFP) format.*

**Syntax**

```c
lapack_int LAPACKE_spftrf (int matrix_layout, char transr, char uplo, lapack_int n, float * a);

lapack_int LAPACKE_dpftrf (int matrix_layout, char transr, char uplo, lapack_int n, double * a);

lapack_int LAPACKE_cpftrf (int matrix_layout, char transr, char uplo, lapack_int n, lapack_complex_float * a);

lapack_int LAPACKE_zpftrf (int matrix_layout, char transr, char uplo, lapack_int n, lapack_complex_double * a);
```
Include Files

• mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, a
Hermitian positive-definite matrix \( A \):

\[
A = U^T U \quad \text{for real data,} \quad A = U^H U \quad \text{for complex data}
\]

\[
A = L^T L \quad \text{for real data,} \quad A = L^H L \quad \text{for complex data}
\]

if \( \text{uplo} = 'U' \)

where \( L \) is a lower triangular matrix and \( U \) is upper triangular.

The matrix \( A \) is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

This is the block version of the algorithm, calling Level 3 BLAS.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

transr

Must be 'N', 'T' (for real data) or 'C' (for complex data).

If \( \text{transr} = 'N' \), the Normal \( \text{transr} \) of RFP \( A \) is stored.

If \( \text{transr} = 'T' \), the Transpose \( \text{transr} \) of RFP \( A \) is stored.

If \( \text{transr} = 'C' \), the Conjugate-Transpose \( \text{transr} \) of RFP \( A \) is stored.

uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of \( A \) is stored:

If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular part of the matrix \( A \).

If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular part of the matrix \( A \).

n

The order of the matrix \( A \); \( n \geq 0 \).

a

Array, size \( (n^2 + n + 1)/2 \). The array \( a \) contains the matrix \( A \) in the RFP format.

Output Parameters

\( a \) is overwritten by the Cholesky factor \( U \) or \( L \), as specified by \( \text{uplo} \) and \( \text{trans} \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix \( A \).
See Also
Matrix Storage Schemes

?pptrf
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite matrix using packed storage.

Syntax

```
lapack_int LAPACKE_spptrf (int matrix_layout, char uplo, lapack_int n, float * ap);
lapack_int LAPACKE_dpptrf (int matrix_layout, char uplo, lapack_int n, double * ap);
lapack_int LAPACKE_cpptrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * ap);
lapack_int LAPACKE_zpptrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * ap);
```

Include Files

- mkl.h

Description

The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite packed matrix \( A \):

\[
A = U^T U \quad \text{for real data}, \quad A = U^H U \quad \text{for complex data}
\]

where \( L \) is a lower triangular matrix and \( U \) is upper triangular.

\[
A = L L^T \quad \text{for real data}, \quad A = L L^H \quad \text{for complex data}
\]

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.

  Indicates whether the upper or lower triangular part of \( A \) is packed in the array \( ap \), and how \( A \) is factored:

  - If \( uplo = 'U' \), the array \( ap \) stores the upper triangular part of the matrix \( A \), and \( A \) is factored as \( U^T U \).
  - If \( uplo = 'L' \), the array \( ap \) stores the lower triangular part of the matrix \( A \); \( A \) is factored as \( L L^T \).

- **n**
  The order of matrix \( A \); \( n \geq 0 \).
Array, size at least max(1, n(n+1)/2). The array \( ap \) contains either the upper or the lower triangular part of the matrix \( A \) (as specified by \( uplo \)) in packed storage (see Matrix Storage Schemes).

**Output Parameters**

\( ap \) Overwritten by the Cholesky factor \( U \) or \( L \), as specified by \( uplo \).

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix \( A \).

**Application Notes**

If \( uplo = 'U' \), the computed factor \( U \) is the exact factor of a perturbed matrix \( A + E \), where

\[
|E| \leq c(n) \epsilon \|U\| \|U^T\|, \quad |e_{ij}| \leq c(n) \epsilon \sqrt{a_{ii}a_{jj}}
\]

\( c(n) \) is a modest linear function of \( n \), and \( \epsilon \) is the machine precision.

A similar estimate holds for \( uplo = 'L' \).

The total number of floating-point operations is approximately \((1/3) n^3\) for real flavors and \((4/3) n^3\) for complex flavors.

After calling this routine, you can call the following routines:

- \( ?pptrs \) to solve \( A^*X = B \)
- \( ?ppcon \) to estimate the condition number of \( A \)
- \( ?pptri \) to compute the inverse of \( A \).

**See Also**

mkl_progress

**Matrix Storage Schemes**

\( ?pbtrf \)

*Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite band matrix.*

**Syntax**

```c
lapack_int LAPACKE_spbtrf (int matrix_layout, char uplo, lapack_int n, lapack_int kd, float * ab, lapack_int ldab);
lapack_int LAPACKE_dpbtrf (int matrix_layout, char uplo, lapack_int n, lapack_int kd, double * ab, lapack_int ldab);
lapack_int LAPACKE_cpbtrf (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_complex_float * ab, lapack_int ldab);
```
lapack_int LAPACKE_zpbtrf (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab);

Include Files
• mkl.h

Description
The routine forms the Cholesky factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix \( A \):
\[
A = U^T U \quad \text{for real data,} \quad A = U^H U \quad \text{for complex data}
\]
\[
A = L L^T \quad \text{for real data,} \quad A = L L^H \quad \text{for complex data}
\]
where \( L \) is a lower triangular matrix and \( U \) is upper triangular.

NOTE
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

\textbf{matrix_layout}

Specifies whether matrix storage layout is row major \texttt{(LAPACK\_ROW\_MAJOR)} or column major \texttt{(LAPACK\_COL\_MAJOR)}.

\textbf{uplo}

Must be \texttt{'U'} or \texttt{'L'}.

Indicates whether the upper or lower triangular part of \( A \) is stored in the array \( ab \), and how \( A \) is factored:
- If \( uplo = 'U' \), the upper triangle of \( A \) is stored.
- If \( uplo = 'L' \), the lower triangle of \( A \) is stored.

\textbf{n}

The order of matrix \( A \); \( n \geq 0 \).

\textbf{kd}

The number of superdiagonals or subdiagonals in the matrix \( A \); \( kd \geq 0 \).

\textbf{ab}

Array, size \( \max(1, ldab \times n) \). The array \( ab \) contains either the upper or the lower triangular part of the matrix \( A \) (as specified by \texttt{uplo}) in band storage (see Matrix Storage Schemes).

\textbf{ldab}

The leading dimension of the array \( ab \). \( (ldab \geq kd + 1) \)

Output Parameters

\textbf{ab}

The upper or lower triangular part of \( A \) (in band storage) is overwritten by the Cholesky factor \( U \) or \( L \), as specified by \texttt{uplo}.

Return Values

This function returns a value \texttt{info}.

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive-definite, and the factorization could not be completed. This may indicate an error in forming the matrix \( A \).
Application Notes

If uplo = 'U', the computed factor $U$ is the exact factor of a perturbed matrix $A + E$, where

$$
|E| \leq c(kd + 1)\varepsilon \left|U^H |U| \right|, \quad |E_{i,j}| \leq c(kd + 1)\varepsilon \sqrt{a_{i,i}a_{j,j}}
$$

$c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision.

A similar estimate holds for uplo = 'L'.

The total number of floating-point operations for real flavors is approximately $n(kd+1)^2$. The number of operations for complex flavors is 4 times greater. All these estimates assume that $kd$ is much less than $n$.

After calling this routine, you can call the following routines:

- ?pbtrs to solve $A^tX = B$
- ?pbcon to estimate the condition number of $A$.

See Also

mkl_progress

Matrix Storage Schemes

?pttrf
Computes the factorization of a symmetric (Hermitian) positive-definite tridiagonal matrix.

Syntax

```c
lapack_int LAPACKE_spttrf( lapack_int n, float* d, float* e );
lapack_int LAPACKE_dpttrf( lapack_int n, double* d, double* e );
lapack_int LAPACKE_cpttrf( lapack_int n, float* d, lapack_complex_float* e );
lapack_int LAPACKE_zpttrf( lapack_int n, double* d, lapack_complex_double* e );
```

Include Files

- mkl.h

Description

The routine forms the factorization of a symmetric positive-definite or, for complex data, Hermitian positive-definite tridiagonal matrix $A$:

- $A = L^*D*L^T$ for real flavors, or
- $A = L^*D*L^H$ for complex flavors,

where $D$ is diagonal and $L$ is unit lower bidiagonal. The factorization may also be regarded as having the form

- $A = U^*D*U$ for real flavors, or $A = U^H*D*U$ for complex flavors, where $U$ is unit upper bidiagonal.

Input Parameters

- $n$ The order of the matrix $A$; $n \geq 0$.
- $d$ Array, dimension ($n$). Contains the diagonal elements of $A$.
- $e$ Array, dimension ($n - 1$). Contains the subdiagonal elements of $A$. 
Output Parameters

d
Overwritten by the $n$ diagonal elements of the diagonal matrix $D$ from the $L^*D*L^T$ (for real flavors) or $L^*D*L^H$ (for complex flavors) factorization of $A$.

e
Overwritten by the $(n - 1)$ sub-diagonal elements of the unit bidiagonal factor $L$ or $U$ from the factorization of $A$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, the leading minor of order $i$ (and therefore the matrix $A$ itself) is not positive-definite; if $i < n$, the factorization could not be completed, while if $i = n$, the factorization was completed, but $d[n - 1] \leq 0$.

?sytrf
Computes the Bunch-Kaufman factorization of a symmetric matrix.

Syntax

lapack_int LAPACKE_ssytrf (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_dsytrf (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_csytrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zsytrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);

Include Files

• mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix $A$ using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- if $uplo='U'$, $A = U^*D*U^T$
- if $uplo='L'$, $A = L^*D*L^T$

where $A$ is the input matrix, $U$ and $L$ are products of permutation and triangular matrices with unit diagonal (upper triangular for $U$ and lower triangular for $L$), and $D$ is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. $U$ and $L$ have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of $D$.

NOTE This routine supports the Progress Routine feature. See Progress Routines section for details.
Input Parameters

matrix_layout
- Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo
- Must be 'U' or 'L'.
- Indicates whether the upper or lower triangular part of A is stored and how A is factored:
  - If uplo = 'U', the array a stores the upper triangular part of the matrix A, and A is factored as $U^TDU^T$.
  - If uplo = 'L', the array a stores the lower triangular part of the matrix A, and A is factored as $LDL^T$.

n
- The order of matrix A; $n \geq 0$.

a
- Array, size max(1, lda*n). The array a contains either the upper or the lower triangular part of the matrix A (see uplo).

lda
- The leading dimension of a; at least max(1, n).

Output Parameters

a
- The upper or lower triangular part of a is overwritten by details of the block-diagonal matrix D and the multipliers used to obtain the factor U (or L).

ipiv
- Array, size at least max(1, n). Contains details of the interchanges and the block structure of D. If $ipiv[i-1] = k > 0$, then $d_{ii}$ is a 1-by-1 block, and the i-th row and column of A was interchanged with the k-th row and column.
  - If uplo = 'U' and $ipiv[i] = ipiv[i-1] = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and i-th row and column of A was interchanged with the m-th row and column.
  - If uplo = 'L' and $ipiv[i] = ipiv[i-1] = -m < 0$, then D has a 2-by-2 block in rows/columns i and $i+1$, and ($i+1$)-th row and column of A was interchanged with the m-th row and column.

Return Values

This function returns a value info.

- If info = 0, the execution is successful.
- If info = -i, parameter i had an illegal value.
- If info = i, $D_{ii}$ is 0. The factorization has been completed, but D is exactly singular. Division by 0 will occur if you use D for solving a system of linear equations.

Application Notes

- The 2-by-2 unit diagonal blocks and the unit diagonal elements of U and L are not stored. The remaining elements of U and L are stored in the corresponding columns of the array a, but additional row interchanges are required to recover U or L explicitly (which is seldom necessary).
- If $ipiv[i-1] = i$ for all $i = 1 \ldots n$, then all off-diagonal elements of U (L) are stored explicitly in the corresponding elements of the array a.
If \( \text{uplo} = 'U' \), the computed factors \( U \) and \( D \) are the exact factors of a perturbed matrix \( A + E \), where
\[
|E| \leq c(n)\varepsilon |U||D| |U^T| |P|
\]
c\( (n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision. A similar estimate holds for the computed \( L \) and \( D \) when \( \text{uplo} = 'L' \).

The total number of floating-point operations is approximately \( (1/3)n^3 \) for real flavors or \( (4/3)n^3 \) for complex flavors.

After calling this routine, you can call the following routines:

- \( ?\text{sytrs} \) to solve \( AX = B \)
- \( ?\text{sycon} \) to estimate the condition number of \( A \)
- \( ?\text{sytri} \) to compute the inverse of \( A \).

If \( \text{uplo} = 'U' \), then \( A = U^TDU^T \), where
\[
U = P(n)^*U(n)^* ... P(k)^*U(k)^* ...,\] that is, \( U \) is a product of terms \( P(k)^*U(k) \), where
- \( k \) decreases from \( n \) to 1 in steps of 1 and 2.
- \( D \) is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks \( D(k) \).
- \( P(k) \) is a permutation matrix as defined by \( \text{ipiv}[k-1] \).
- \( U(k) \) is a unit upper triangular matrix, such that if the diagonal block \( D(k) \) is of order \( s \) (\( s = 1 \) or 2), then
\[
U(k) = \begin{bmatrix}
I & V & 0 & & & & & \\
0 & I & 0 & & & & & \\
0 & 0 & I & & & & & \\
& & & I & 0 & & & \\
& & & & I & 0 & & \\
& & & & & I & 0 & \\
& & & & & & I & 0 \\
& & & & & & & I
\end{bmatrix}
\]
If \( s = 1 \), \( D(k) \) overwrites \( A(k,k) \), and \( V \) overwrites \( A(1:k-1,k) \).

If \( s = 2 \), the upper triangle of \( D(k) \) overwrites \( A(k-1,k-1), A(k-1,k) \) and \( A(k,k) \), and \( V \) overwrites \( A(1:k-2,k-1:k) \).

If \( \text{uplo} = 'L' \), then \( A = L^TDL^T \), where
\[
L = P(1)^*L(1)^* ... P(k)^*L(k)^* ...,\] that is, \( L \) is a product of terms \( P(k)^*L(k) \), where
- \( k \) increases from 1 to \( n \) in steps of 1 and 2.
- \( D \) is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks \( D(k) \).
- \( P(k) \) is a permutation matrix as defined by \( \text{ipiv}(k) \).
- \( L(k) \) is a unit lower triangular matrix, such that if the diagonal block \( D(k) \) is of order \( s \) (\( s = 1 \) or 2), then
If $s = 1$, $D(k)$ overwrites $A(k,k)$, and $v$ overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $D(k)$ overwrites $A(k,k)$, $A(k+1,k)$, and $A(k+1,k+1)$, and $v$ overwrites $A(k+2:n,k:k+1)$.

**See Also**

mkl_progress

**Matrix Storage Schemes**

?sytrf_aa

*Computes the factorization of a symmetric matrix using Aasen's algorithm.*

```c
lapack_int LAPACKE_ssytrf_aa (int matrix_layout, char uplo, lapack_int n, float * A, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_dsytrf_aa (int matrix_layout, char uplo, lapack_int n, double * A, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_csytrf_aa (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * A, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zsytrf_aa (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * A, lapack_int lda, lapack_int * ipiv);
```

**Description**

?sytrf_aa computes the factorization of a symmetric matrix $A$ using Aasen's algorithm. The form of the factorization is $A = U^*T*U$ or $A = L^*T*L$ where $U$ (or $L$) is a product of permutation and unit upper (lower) triangular matrices, and $T$ is a complex symmetric tridiagonal matrix.

This is the blocked version of the algorithm, calling Level 3 BLAS.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  
  - `'U'`: The upper triangle of $A$ is stored.
  - `'L'`: The lower triangle of $A$ is stored.

- **n**
  
  The order of the matrix $A$. $n \geq 0$.

- **A**
  
  Array of size max(1, lda*n). The array $A$ contains either the upper or the lower triangular part of the matrix $A$ (see uplo).

- **lda**
  
  The leading dimension of the array $A$. 

\[
L(k) = \begin{pmatrix}
  I & 0 & 0 & k-1 \\
  0 & I & 0 & s \\
  0 & v & I & n-k-s+1 \\
  k-1 & s & n-k-s+1
\end{pmatrix}
\]
Output Parameters

\( A \)

On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of \( A \) just below (or above) the diagonals, and \( L \) is stored below (or above) the subdiagonals, when \( \text{uplo} \) is 'L' (or 'U').

\( \text{ipiv} \)

Array of size \( n \). On exit, it contains the details of the interchanges; that is, the row and column \( k \) of \( A \) were interchanged with the row and column \( \text{ipiv}(k) \).

Return Values

This function returns a value \( \text{info} \).

\( = 0 \): Successful exit.

\( < 0 \): If \( \text{info} = -i \), the \( i \)th argument had an illegal value.

\( > 0 \): If \( \text{info} = i \), \( D(i,i) \) is exactly zero. The factorization has been completed, but the block diagonal matrix \( D \) is exactly singular, and division by zero will occur if it is used to solve a system of equations.

\textbf{sytrf_rook}

Computes the bounded Bunch-Kaufman factorization of a symmetric matrix.

Syntax

```c
lapack_int LAPACKE_ssytrf_rook (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_dsytrf_rook (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_csytrf_rook (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);
lapack_int LAPACKE_zsytrf_rook (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix \( A \) using the bounded Bunch-Kaufman ("rook") diagonal pivoting method. The form of the factorization is:

\[
\begin{align*}
\text{if } \text{uplo} = 'U', & \quad A = U^* D U^T \\
\text{if } \text{uplo} = 'L', & \quad A = L^* D L^T,
\end{align*}
\]

where \( A \) is the input matrix, \( U \) and \( L \) are products of permutation and triangular matrices with unit diagonal (upper triangular for \( U \) and lower triangular for \( L \)), and \( D \) is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. \( U \) and \( L \) have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of \( D \).

Input Parameters

\( \text{matrix_layout} \)

Specifies whether matrix storage layout for array \( b \) is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:

If $\text{uplo} = 'U'$, the array $a$ stores the upper triangular part of the matrix $A$, and $A$ is factored as $U*D*U^T$.

If $\text{uplo} = 'L'$, the array $a$ stores the lower triangular part of the matrix $A$, and $A$ is factored as $L*D*L^T$.

$n$

The order of matrix $A$; $n \geq 0$.

$a$

Array, size $lda*n$. The array $a$ contains either the upper or the lower triangular part of the matrix $A$ (see $\text{uplo}$).

$lda$

The leading dimension of $a$; at least $\max(1, n)$.

Output Parameters

The upper or lower triangular part of $a$ is overwritten by details of the block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$).

$ipiv$

If $\text{ipiv}(k) > 0$, then rows and columns $k$ and $\text{ipiv}(k)$ were interchanged and $D_{k,k}$ is a 1-by-1 diagonal block.

If $\text{uplo} = 'U'$ and $\text{ipiv}(k) < 0$ and $\text{ipiv}(k - 1) < 0$, then rows and columns $k$ and $-\text{ipiv}(k)$ were interchanged, rows and columns $k - 1$ and $-\text{ipiv}(k - 1)$ were interchanged, and $D_{k-1:k, k-1:k}$ is a 2-by-2 diagonal block.

If $\text{uplo} = 'L'$ and $\text{ipiv}(k) < 0$ and $\text{ipiv}(k + 1) < 0$, then rows and columns $k$ and $-\text{ipiv}(k)$ were interchanged, rows and columns $k + 1$ and $-\text{ipiv}(k + 1)$ were interchanged, and $D_{k:k+1, k:k+1}$ is a 2-by-2 diagonal block.

Return Values

This function returns a value $\text{info}$.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.

If $\text{info} = i$, $D_{ii}$ is 0. The factorization has been completed, but $D$ is exactly singular. Division by 0 will occur if you use $D$ for solving a system of linear equations.

Application Notes

The total number of floating-point operations is approximately $\frac{1}{3}n^3$ for real flavors or $\frac{4}{3}n^3$ for complex flavors.

After calling this routine, you can call the following routines:

$\text{?sytrs_rook}$ to solve $A*X = B$

$\text{?sycon_rook}$ (Fortran only) to estimate the condition number of $A$

$\text{?sytri_rook}$ (Fortran only) to compute the inverse of $A$.
If `uplo` = 'U', then $A = U*\mathcal{D}*U'$, where

$$U = P(n)*U(n)* \ldots *P(k)*U(k)*\ldots,$$

that is, $U$ is a product of terms $P(k)*U(k)$, where

- $k$ decreases from $n$ to 1 in steps of 1 and 2.
- $\mathcal{D}$ is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $\mathcal{D}(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv[k-1].`
- $U(k)$ is a unit upper triangular matrix, such that if the diagonal block $\mathcal{D}(k)$ is of order $s$ ($s = 1$ or 2), then

$$U(k) = \begin{pmatrix}
  I & v & 0 \\
  0 & I & 0 \\
  0 & 0 & I \\
\end{pmatrix}
\begin{pmatrix}
  k-s & s & n-k \\
\end{pmatrix}$$

If $s = 1$, $\mathcal{D}(k)$ overwrites $A(k,k)$, and $v$ overwrites $A(1:k-1,k)$.

If $s = 2$, the upper triangle of $\mathcal{D}(k)$ overwrites $A(k-1,k-1), A(k-1,k)$ and $A(k,k)$, and $v$ overwrites $A(1:k-2,k-1:k)$.

If `uplo` = 'L', then $A = L*\mathcal{D}*L'$, where

$$L = P(1)*L(1)* \ldots *P(k)*L(k)*\ldots,$$

that is, $L$ is a product of terms $P(k)*L(k)$, where

- $k$ increases from 1 to $n$ in steps of 1 and 2.
- $\mathcal{D}$ is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks $\mathcal{D}(k)$.
- $P(k)$ is a permutation matrix as defined by `ipiv(k).`
- $L(k)$ is a unit lower triangular matrix, such that if the diagonal block $\mathcal{D}(k)$ is of order $s$ ($s = 1$ or 2), then

$$L(k) = \begin{pmatrix}
  I & 0 & 0 \\
  0 & I & 0 \\
  0 & v & I \\
\end{pmatrix}
\begin{pmatrix}
  k-1 & s & n-k-s+1 \\
\end{pmatrix}$$

If $s = 1$, $\mathcal{D}(k)$ overwrites $A(k,k)$, and $v$ overwrites $A(k+1:n,k)$.

If $s = 2$, the lower triangle of $\mathcal{D}(k)$ overwrites $A(k,k), A(k+1,k)$, and $A(k+1,k+1)$, and $v$ overwrites $A(k+2:n,k+1)$.

**See Also**

**Matrix Storage Schemes**

?-sytrf_rk

*Computes the factorization of a real or complex symmetric indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).*
lapack_int LAPACKE_ssytrf_rk (int matrix_layout, char uplo, lapack_int n, float * A, lapack_int lda, float * e, lapack_int * ipiv);

lapack_int LAPACKE_dsytrf_rk (int matrix_layout, char uplo, lapack_int n, double * A, lapack_int lda, double * e, lapack_int * ipiv);

lapack_int LAPACKE_csytrf_rk (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int * ipiv);

lapack_int LAPACKE_zsytrf_rk (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int * ipiv);

Description

?sytrf_rk computes the factorization of a real or complex symmetric matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: A = P*U*D*(U^T)(P^T) or A = P*L*D*(L^T)(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^T (or L^T) is the transpose of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:
- = 'U': Upper triangular
- = 'L': Lower triangular

n

The order of the matrix A. n ≥ 0.

A

Array of size max(1, lda*n). On entry, the symmetric matrix A. If uplo = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If uplo = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

The leading dimension of the array A.

Output Parameters

A

On exit, contains:
- Only diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(k,k) = A(k,k); (superdiagonal (or subdiagonal) elements of D are stored on exit in array e).
- If uplo = 'U', factor U in the superdiagonal part of A. If uplo = 'L', factor L in the subdiagonal part of A.

e

Array of size n. On exit, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If uplo = 'U', e(i) = D(i-1,i), i=2:N, and e(1) is set to 0. If uplo = 'L', e(i) = D(i+1,i), i=1:N-1, and e(n) is set to 0.
NOTE For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is set to 0 in both the $\text{uplo} = 'U'$ and $\text{uplo} = 'L'$ cases.

Array of size $n$. $\text{ipiv}$ describes the permutation matrix $P$ in the factorization of matrix $A$ as follows: The absolute value of $\text{ipiv}(k)$ represents the index of the row and column that were interchanged with the $k^{th}$ row and column. The value of $\text{uplo}$ describes the order in which the interchanges were applied. Also, the sign of $\text{ipiv}$ represents the block structure of the symmetric block diagonal matrix $D$ with 1-by-1 or 2-by-2 diagonal blocks, which correspond to 1 or 2 interchanges at each factorization step. If $\text{uplo} = 'U'$ (in factorization order, $k$ decreases from $n$ to 1):

1. A single positive entry $\text{ipiv}(k) > 0$ means that $D(k,k)$ is a 1-by-1 diagonal block. If $\text{ipiv}(k) \neq k$, rows and columns $k$ and $\text{ipiv}(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $\text{ipiv}(k) = k$, no interchange occurred.

2. A pair of consecutive negative entries $\text{ipiv}(k) < 0$ and $\text{ipiv}(k-1).< 0$ means that $D(k-1:k,k-1:k)$ is a 2-by-2 diagonal block. (Note that negative entries in $\text{ipiv}$ appear only in pairs.)
   - If $-\text{ipiv}(k) \neq k$, rows and columns $k$ and $-\text{ipiv}(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $-\text{ipiv}(k) = k$, no interchange occurred.
   - If $-\text{ipiv}(k-1) \neq k-1$, rows and columns $k-1$ and $-\text{ipiv}(k-1)$ were interchanged in the matrix $A(1:N,1:N)$. If $-\text{ipiv}(k-1) = k-1$, no interchange occurred.

3. In both cases 1 and 2, always $\text{ABS}(\text{ipiv}(k)) \leq k$.

NOTE Any entry $\text{ipiv}(k)$ is always nonzero on output.

If $\text{uplo} = 'L'$ (in factorization order, $k$ increases from 1 to $n$):

1. A single positive entry $\text{ipiv}(k) > 0$ means that $D(k,k)$ is a 1-by-1 diagonal block. If $\text{ipiv}(k) \neq k$, rows and columns $k$ and $\text{ipiv}(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $\text{ipiv}(k) = k$, no interchange occurred.

2. A pair of consecutive negative entries $\text{ipiv}(k) < 0$ and $\text{ipiv}(k+1) < 0$ means that $D(k:k+1,k:k+1)$ is a 2-by-2 diagonal block. (Note that negative entries in $\text{ipiv}$ appear only in pairs.)
   - If $-\text{ipiv}(k) \neq k$, rows and columns $k$ and $-\text{ipiv}(k)$ were interchanged in the matrix $A(1:N,1:N)$. If $-\text{ipiv}(k) = k$, no interchange occurred.
   - If $-\text{ipiv}(k+1) \neq k+1$, rows and columns $k-1$ and $-\text{ipiv}(k-1)$ were interchanged in the matrix $A(1:N,1:N)$. If $-\text{ipiv}(k+1) = k+1$, no interchange occurred.

3. In both cases 1 and 2, always $\text{ABS}(\text{ipiv}(k)) \geq k$.

NOTE Any entry $\text{ipiv}(k)$ is always nonzero on output.
Return Values
This function returns a value info.

= 0: Successful exit.
< 0: If info = -k, the kth argument had an illegal value.
> 0: If info = k, the matrix A is singular. If uplo = 'U', column k in the upper triangular part of A contains all zeros. If uplo = 'L', column k in the lower triangular part of A contains all zeros. Therefore, D(k,k) is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?hetrf
Computes the Bunch-Kaufman factorization of a complex Hermitian matrix.

Syntax
lapack_int LAPACKE_chetrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , lapack_int * ipiv );
lapack_int LAPACKE_zhetrf (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , lapack_int * ipiv );

Include Files
 • mkl.h

Description
The routine computes the factorization of a complex Hermitian matrix A using the Bunch-Kaufman diagonal pivoting method:

if uplo='U', A = U*D*UH
if uplo='L', A = L*D*LH,
where A is the input matrix, U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

NOTE
This routine supports the Progress Routine feature. See Progress Routines section for details.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo
Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored and how A is factored:
If uplo = 'U', the array a stores the upper triangular part of the matrix A, and A is factored as U*D*UH.
If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular part of the matrix \( A \), and \( A \) is factored as \( L \cdot D \cdot L^T \).

\( n \)

The order of matrix \( A \); \( n \geq 0 \).

\( a \)

Array, size \( \max(1, \text{lda} \cdot n) \).

The array \( a \) contains the upper or the lower triangular part of the matrix \( A \) (see \( \text{uplo} \)).

\( \text{lda} \)

The leading dimension of \( a \); at least \( \max(1, n) \).

### Output Parameters

The upper or lower triangular part of \( a \) is overwritten by details of the block-diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) (or \( L \)).

\( \text{ipiv} \)

Array, size at least \( \max(1, n) \). Contains details of the interchanges and the block structure of \( D \). If \( \text{ipiv}[i-1] = k > 0 \), then \( d_{ii} \) is a 1-by-1 block, and the \( i \)-th row and column of \( A \) was interchanged with the \( k \)-th row and column.

If \( \text{uplo} = 'U' \) and \( \text{ipiv}[i] = \text{ipiv}[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \( i \)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

If \( \text{uplo} = 'L' \) and \( \text{ipiv}[i] = \text{ipiv}[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \((i+1)\)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

### Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), \( d_{ii} \) is 0. The factorization has been completed, but \( D \) is exactly singular. Division by 0 will occur if you use \( D \) for solving a system of linear equations.

### Application Notes

This routine is suitable for Hermitian matrices that are not known to be positive-definite. If \( A \) is in fact positive-definite, the routine does not perform interchanges, and no 2-by-2 diagonal blocks occur in \( D \).

The 2-by-2 unit diagonal blocks and the unit diagonal elements of \( U \) and \( L \) are not stored. The remaining elements of \( U \) and \( L \) are stored in the corresponding columns of the array \( a \), but additional row interchanges are required to recover \( U \) or \( L \) explicitly (which is seldom necessary).

If \( \text{ipiv}[i-1] = i \) for all \( i = 1 \ldots n \), then all off-diagonal elements of \( U \) (\( L \)) are stored explicitly in the corresponding elements of the array \( a \).

If \( \text{uplo} = 'U' \), the computed factors \( U \) and \( D \) are the exact factors of a perturbed matrix \( A + E \), where

\[
|E| \leq c(n)\varepsilon|U||D||U^T|P^T
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

A similar estimate holds for the computed \( L \) and \( D \) when \( \text{uplo} = 'L' \).
The total number of floating-point operations is approximately \((4/3)n^3\).

After calling this routine, you can call the following routines:

- \(?hetrs\) to solve \(A^*X = B\)
- \(?hecon\) to estimate the condition number of \(A\)
- \(?hetri\) to compute the inverse of \(A\).

**See Also**

mkl_progress

**Matrix Storage Schemes**

- \(?hetrf_aa\)
  
  **Computes the factorization of a complex hermitian matrix using Aasen's algorithm.**

  ```c
  LAPACK_DECL lapack_int LAPACKE_chetrf_aa (int matrix_layout, char uplo, lapack_int n,
  lapack_complex_float * a, lapack_int lda, lapack_int * ipiv);
  ```

  ```c
  LAPACK_DECL lapack_int LAPACKE_zhetrf_aa (int matrix_layout, char uplo, lapack_int n,
  lapack_complex_double * a, lapack_int lda, lapack_int * ipiv);
  ```

**Description**

\(?hetrf_aa\) computes the factorization of a complex Hermitian matrix \(A\) using Aasen's algorithm. The form of the factorization is \(A = U^*T^*U^H\) or \(A = L^*T^*L^H\) where \(U\) (or \(L\)) is a product of permutation and unit upper (lower) triangular matrices, and \(T\) is a Hermitian tridiagonal matrix. This is the blocked version of the algorithm, calling Level 3 BLAS.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (\(LAPACK_ROW_MAJOR\)) or column major (\(LAPACK_COL_MAJOR\)).

- **uplo**
  
  \(= 'U'\): Upper triangle of \(A\) is stored; \(= 'L'\): Lower triangle of \(a\) is stored.

- **n**
  
  The order of the matrix \(A\). \(n \geq 0\).

- **a**
  
  Array of size \(lda*n\). On entry, the Hermitian matrix \(A\).

  If \(uplo = 'U'\), the leading \(n\)-by-\(n\) upper triangular part of \(a\) contains the upper triangular part of the matrix \(A\), and the strictly lower triangular part of \(a\) is not referenced.

  If \(uplo = 'L'\), the leading \(n\)-by-\(n\) lower triangular part of \(a\) contains the lower triangular part of the matrix \(A\), and the strictly upper triangular part of \(a\) is not referenced.

- **lda**
  
  The leading dimension of the array \(a\). \(lda \geq max(1,n)\).

- **lwork**
  
  See **Syntax - Workspace**. The length of \(work\). \(lwork \geq 2*n\). For optimum performance \(lwork \geq n*(1 + nb)\), where \(nb\) is the optimal block size. If \(lwork = -1\), then a workspace query is assumed; the routine only calculates the optimal size of the \(work\) array, returns this value as the first entry of the \(work\) array, and no error message related to \(lwork\) is issued by \(xerbla\).
Output Parameters

\textbf{a} 
On exit, the tridiagonal matrix is stored in the diagonals and the subdiagonals of \(a\) just below (or above) the diagonals, and \(L\) is stored below (or above) the subdiagonals, when \textit{uplo} is 'L' (or 'U').

\textbf{ipiv} 
array, dimension (\(n\)) On exit, it contains the details of the interchanges: the row and column \(k\) of \(a\) were interchanged with the row and column \(\text{ipiv}[k]\).

\textbf{work} 
See Syntax - Workspace. Array of size (max(1, \textit{lwork})). On exit, if \textit{info} = 0, \textit{work}[0] returns the optimal \textit{lwork}.

Return Values

This function returns a value \textit{info}.

If \textit{info} = 0: successful exit < 0: if \textit{info} = -i, the \(i\)-th argument had an illegal value,

If \textit{info} > 0: if \textit{info} = \(i\), \(D_{ii}\) is exactly zero. The factorization has been completed, but the block diagonal matrix \(D\) is exactly singular, and division by zero will occur if it is used to solve a system of equations.

Syntax - Workspace

Use this interface if you want to explicitly provide the workspace array.

LAPACK DECL lapack_int LAPACKE_chetrf_aa_work (int \textit{matrix_layout}, char \textit{uplo}, lapack_int \textit{n}, lapack_complex_float * \textit{a}, lapack_int \textit{lda}, lapack_int * \textit{ipiv}, lapack_complex_float * \textit{work}, lapack_int \textit{lwork });

LAPACK DECL lapack_int LAPACKE_zhetrf_aa_work (int \textit{matrix_layout}, char \textit{uplo}, lapack_int \textit{n}, lapack_complex_double * \textit{a}, lapack_int \textit{lda}, lapack_int * \textit{ipiv}, lapack_complex_double * \textit{work}, lapack_int \textit{lwork });

\textbf{?hetrf_rook} 
\textit{Computes the bounded Bunch-Kaufman factorization of a complex Hermitian matrix.}

Syntax

\texttt{lapack_int LAPACKE_chetrf_rook (int \textit{matrix_layout}, char \textit{uplo}, lapack_int \textit{n}, lapack_complex_float * \textit{a}, lapack_int \textit{lda}, lapack_int * \textit{ipiv});}

\texttt{lapack_int LAPACKE_zhetrf_rook (int \textit{matrix_layout}, char \textit{uplo}, lapack_int \textit{n}, lapack_complex_double * \textit{a}, lapack_int \textit{lda}, lapack_int * \textit{ipiv});}

Include Files

\bullet \texttt{mkl.h}

Description

The routine computes the factorization of a complex Hermitian matrix \(A\) using the bounded Bunch-Kaufman diagonal pivoting method:

\begin{align*}
\text{if } \textit{uplo} = 'U', & A = U \times D \times U^H \\
\text{if } \textit{uplo} = 'L', & A = L \times D \times L^H,
\end{align*}

where \(A\) is the input matrix, \(U\) (or \(L\) ) is a product of permutation and unit upper ( or lower) triangular matrices, and \(D\) is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.
Input Parameters

matrix_layout

Specifies whether matrix storage layout for array b is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is stored:
If uplo = 'U', the array a stores the upper triangular part of the matrix A.
If uplo = 'L', the array a stores the lower triangular part of the matrix A.

n

The order of matrix A; n ≥ 0.

a

Array a, size (lda*n)
The array a contains the upper or the lower triangular part of the matrix A (see uplo).
If uplo = 'U', the leading n-by-n upper triangular part of a contains the upper triangular part of the matrix A, and the strictly lower triangular part of a is not referenced. If uplo = 'L', the leading n-by-n lower triangular part of a contains the lower triangular part of the matrix A, and the strictly upper triangular part of a is not referenced.

lda

The leading dimension of a; at least max(1, n).

Output Parameters

a

The block diagonal matrix D and the multipliers used to obtain the factor U or L (see Application Notes for further details).

ipiv

- If uplo = 'U':
  If ipiv(k) > 0, then rows and columns k and ipiv(k) were interchanged and D_{k,k} is a 1-by-1 diagonal block.
  If ipiv(k) < 0 and ipiv(k - 1) < 0, then rows and columns k and -ipiv(k) were interchanged and rows and columns k - 1 and -ipiv(k - 1) were interchanged, D_{k-1:k,k-1:k} is a 2-by-2 diagonal block.
- If uplo = 'L':
  If ipiv(k) > 0, then rows and columns k and ipiv(k) were interchanged and D_{k,k} is a 1-by-1 diagonal block.
  If ipiv(k) < 0 and ipiv(k + 1) < 0, then rows and columns k and -ipiv(k) were interchanged and rows and columns k + 1 and -ipiv(k + 1) were interchanged, D_{k:k+1,k:k+1} is a 2-by-2 diagonal block.

Return Values

This function returns a value info.
If info = 0, the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = i \), \( D_{ii} \) is exactly 0. The factorization has been completed, but the block diagonal matrix \( D \) is exactly singular, and division by 0 will occur if you use \( D \) for solving a system of linear equations.

### Application Notes

If \( \text{uplo} = 'U' \), then \( A = U*D*U^H \), where

\[
U = P(n)*U(n)* ... *P(k)U(k)* ..., \]

i.e., \( U \) is a product of terms \( P(k)*U(k) \), where \( k \) decreases from \( n \) to 1 in steps of 1 or 2, and \( D \) is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks \( D(k) \). \( P(k) \) is a permutation matrix as defined by \( \text{ipiv}(k) \), and \( U(k) \) is a unit upper triangular matrix, such that if the diagonal block \( D(k) \) is of order \( s \) (\( s = 1 \) or 2), then

\[
U(k) = \begin{bmatrix}
    I & v & 0 \\
    s & I & 0 \\
    n-k & 0 & I
\end{bmatrix}
\]

If \( s = 1 \), \( D(k) \) overwrites \( A(k,k) \), and \( v \) overwrites \( A(1:k-1,k) \).

If \( s = 2 \), the upper triangle of \( D(k) \) overwrites \( A(k-1,k-1) \), \( A(k-1,k) \), and \( A(k,k) \), and \( v \) overwrites \( A(1:k-2,k-1:k) \).

If \( \text{uplo} = 'L' \), then \( A = L*D*L^H \), where

\[
L = P(1)*L(1)* ... *P(k)*L(k)* ..., \]

i.e., \( L \) is a product of terms \( P(k)*L(k) \), where \( k \) increases from 1 to \( n \) in steps of 1 or 2, and \( D \) is a block diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks \( D(k) \). \( P(k) \) is a permutation matrix as defined by \( \text{ipiv}(k) \), and \( L(k) \) is a unit lower triangular matrix, such that if the diagonal block \( D(k) \) is of order \( s \) (\( s = 1 \) or 2), then

\[
L(k) = \begin{bmatrix}
    I & 0 & 0 \\
    s & I & 0 \\
    n-k-s+1 & 0 & v & I
\end{bmatrix}
\]

If \( s = 1 \), \( D(k) \) overwrites \( A(k,k) \), and \( v \) overwrites \( A(k+1:n,k) \).

If \( s = 2 \), the lower triangle of \( D(k) \) overwrites \( A(k,k) \), \( A(k+1,k) \), and \( A(k+1,k+1) \), and \( v \) overwrites \( A(k+2:n,k+1) \).

### See Also

mkl_progress

### Matrix Storage Schemes

**?hetrf_rk**

Computes the factorization of a complex Hermitian indefinite matrix using the bounded Bunch-Kaufman (rook) diagonal pivoting method (BLAS3 blocked algorithm).

```c
lapack_int LAPACKE_chetrf_rk (int matrix_layout, char uplo, lapack_int n,
  lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int * ipiv);

lapack_int LAPACKE_zhetrf_rk (int matrix_layout, char uplo, lapack_int n,
  lapack_complex_double * A, lapack_int lda, lapack_complex_double * e, lapack_int * ipiv);
```
Description

?hetrf_rk computes the factorization of a complex Hermitian matrix A using the bounded Bunch-Kaufman (rook) diagonal pivoting method: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This is the blocked version of the algorithm, calling Level 3 BLAS.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Specifies whether the upper or lower triangular part of the Hermitian matrix A is stored:

- = 'U': Upper triangular.
- = 'L': Lower triangular.

n

The order of the matrix A, n ≥ 0.

A

Array of size max(1, lda*n). On entry, the Hermitian matrix A. If uplo = 'U': The leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If uplo = 'L': The leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

lda

The leading dimension of the array A.

Output Parameters

A

On exit, contains:

- Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, D(k,k) = A(k,k). Superdiagonal (or subdiagonal) elements of D are stored on exit in array e.

- and -
- If uplo = 'U', factor U in the superdiagonal part of A. If uplo = 'L', factor L in the subdiagonal part of A.

e

Array of size n. On exit, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If uplo = 'U', e(i) = D(i-1,i), i=2:N, and e(1) is set to 0. If uplo = 'L', e(i) = D(i+1,i), i=1:N-1, and e(n) is set to 0.

NOTE For 1-by-1 diagonal block D(k), where 1 ≤ k ≤ n, the element e[k-1] is set to 0 in both the uplo = 'U' and uplo = 'L' cases.

ipiv

Array of size n. ipiv describes the permutation matrix P in the factorization of matrix A as follows: The absolute value of ipiv[k-1] represents the index of row and column that were interchanged with the kth row and column. The value of uplo describes the order in which the interchanges
were applied. Also, the sign of \( ipiv \) represents the block structure of the Hermitian block diagonal matrix \( D \) with 1-by-1 or 2-by-2 diagonal blocks that correspond to 1 or 2 interchanges at each factorization step. If \( \text{uplo} = 'U' \) (in factorization order, \( k \) decreases from \( n \) to 1):

1. A single positive entry \( ipiv(k) > 0 \) means that \( D(k,k) \) is a 1-by-1 diagonal block. If \( ipiv(k) \neq k \), rows and columns \( k \) and \( ipiv(k) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( ipiv(k) = k \), no interchange occurred.
2. A pair of consecutive negative entries \( ipiv(k) < 0 \) and \( ipiv(k-1) < 0 \) means that \( D(k-1:k-1,k:k) \) is a 2-by-2 diagonal block. (Note that negative entries in \( ipiv \) appear only in pairs.)
   - If \( -ipiv(k) \neq k \), rows and columns \( k \) and \( -ipiv(k) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( -ipiv(k) = k \), no interchange occurred.
   - If \( -ipiv(k-1) \neq k-1 \), rows and columns \( k-1 \) and \( -ipiv(k-1) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( -ipiv(k-1) = k-1 \), no interchange occurred.
3. In both cases 1 and 2, always \( \text{ABS}( ipiv(k) ) \leq k \).

**NOTE** Any entry \( ipiv(k) \) is always nonzero on output.

If \( \text{uplo} = 'L' \) (in factorization order, \( k \) increases from 1 to \( n \)):

1. A single positive entry \( ipiv(k) > 0 \) means that \( D(k,k) \) is a 1-by-1 diagonal block. If \( ipiv(k) \neq k \), rows and columns \( k \) and \( ipiv(k) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( ipiv(k) = k \), no interchange occurred.
2. A pair of consecutive negative entries \( ipiv(k) < 0 \) and \( ipiv(k+1) < 0 \) means that \( D(k:k+1,k:k+1) \) is a 2-by-2 diagonal block. (Note that negative entries in \( ipiv \) appear only in pairs.)
   - If \( -ipiv(k) \neq k \), rows and columns \( k \) and \( -ipiv(k) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( -ipiv(k) = k \), no interchange occurred.
   - If \( -ipiv(k+1) \neq k+1 \), rows and columns \( k-1 \) and \( -ipiv(k-1) \) were interchanged in the matrix \( A(1:N,1:N) \). If \( -ipiv(k+1) = k+1 \), no interchange occurred.
3. In both cases 1 and 2, always \( \text{ABS}( ipiv(k) ) \geq k \).

**NOTE** Any entry \( ipiv(k) \) is always nonzero on output.

**Return Values**
This function returns a value \( info \).

\( = 0 \): Successful exit.

\( < 0 \): If \( info = -k \), the \( k \)th argument had an illegal value.
> 0: If info = k, the matrix A is singular. If uplo = 'U', the column k in the upper triangular part of A contains all zeros. If uplo = 'L', the column k in the lower triangular part of A contains all zeros. Therefore D(k,k) is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sptrf
Computes the Bunch-Kaufman factorization of a symmetric matrix using packed storage.

Syntax

```c
lapack_int LAPACKE_ssptrf (int matrix_layout, char uplo, lapack_int n, float * ap, lapack_int * ipiv);

lapack_int LAPACKE_dsptrf (int matrix_layout, char uplo, lapack_int n, double * ap, lapack_int * ipiv);

lapack_int LAPACKE_csptrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * ap, lapack_int * ipiv);

lapack_int LAPACKE_zsptrf (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * ap, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the factorization of a real/complex symmetric matrix A stored in the packed format using the Bunch-Kaufman diagonal pivoting method. The form of the factorization is:

- If uplo='U', A = U*D*Uᵀ
- If uplo='L', A = L*D*Lᵀ,

where U and L are products of permutation and triangular matrices with unit diagonal (upper triangular for U and lower triangular for L), and D is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. U and L have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of D.

**NOTE**
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of A is packed in the array ap and how A is factored:
  - If uplo = 'U', the array ap stores the upper triangular part of the matrix A, and A is factored as U*D*Uᵀ.
  - If uplo = 'L', the array ap stores the lower triangular part of the matrix A, and A is factored as L*D*Lᵀ.
The order of matrix $A$; $n \geq 0$.

Array, size at least $\max(1, n(n+1)/2)$. The array $ap$ contains the upper or the lower triangular part of the matrix $A$ (as specified by $uplo$) in packed storage (see Matrix Storage Schemes).

Output Parameters

The upper or lower triangle of $A$ (as specified by $uplo$) is overwritten by details of the block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$).

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of $D$. If $ipiv[i-1] = k > 0$, then $d_{ii}$ is a 1-by-1 block, and the $i$-th row and column of $A$ was interchanged with the $k$-th row and column.

If $uplo = 'U'$ and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $i$-th row and column of $A$ was interchanged with the $m$-th row and column.

If $uplo = 'L'$ and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $(i+1)$-th row and column of $A$ was interchanged with the $m$-th row and column.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, $d_{ii}$ is 0. The factorization has been completed, but $D$ is exactly singular. Division by 0 will occur if you use $D$ for solving a system of linear equations.

Application Notes

The 2-by-2 unit diagonal blocks and the unit diagonal elements of $U$ and $L$ are not stored. The remaining elements of $U$ and $L$ overwrite elements of the corresponding columns of the array $ap$, but additional row interchanges are required to recover $U$ or $L$ explicitly (which is seldom necessary).

If $ipiv(i) = i$ for all $i = 1 \ldots n$, then all off-diagonal elements of $U$ ($L$) are stored explicitly in packed form.

If $uplo = 'U'$, the computed factors $U$ and $D$ are the exact factors of a perturbed matrix $A + E$, where

$$|E| \leq c(n)\|P\|\|U\|\|D\|\|U^T\|P^T$$

$c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision. A similar estimate holds for the computed $L$ and $D$ when $uplo = 'L'$.

The total number of floating-point operations is approximately $(1/3)n^3$ for real flavors or $(4/3)n^3$ for complex flavors.

After calling this routine, you can call the following routines:

- ?sptrs to solve $A^*X = B$
- ?spcon to estimate the condition number of $A$
- ?sptri to compute the inverse of $A$.  

See Also
mkl_progress

Matrix Storage Schemes

?hptrf
Computes the Bunch-Kaufman factorization of a complex Hermitian matrix using packed storage.

Syntax

```c
lapack_int LAPACKE_chptrf (int matrix_layout, char uplo, lapack_int n, 
lapack_complex_float * ap, lapack_int * ipiv);
lapack_int LAPACKE_zhptrf (int matrix_layout, char uplo, lapack_int n, 
lapack_complex_double * ap, lapack_int * ipiv);
```

Include Files

- mkl.h

Description

The routine computes the factorization of a complex Hermitian packed matrix $A$ using the Bunch-Kaufman diagonal pivoting method:

- If $\text{uplo} = 'U'$, $A = U*D*U^H$
- If $\text{uplo} = 'L'$, $A = L*D*L^H$,

where $A$ is the input matrix, $U$ and $L$ are products of permutation and triangular matrices with unit diagonal (upper triangular for $U$ and lower triangular for $L$), and $D$ is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. $U$ and $L$ have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of $D$.

**NOTE**
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of $A$ is packed and how $A$ is factored:
  - If $\text{uplo} = 'U'$, the array $ap$ stores the upper triangular part of the matrix $A$, and $A$ is factored as $U*D*U^H$.
  - If $\text{uplo} = 'L'$, the array $ap$ stores the lower triangular part of the matrix $A$, and $A$ is factored as $L*D*L^H$.

- **n**
  The order of matrix $A$; $n \geq 0$. 
**Ap**

Array, size at least \( \max(1, n(n+1)/2) \). The array \( \text{ap} \) contains the upper or the lower triangular part of the matrix \( A \) (as specified by \( \text{uplo} \)) in packed storage (see Matrix Storage Schemes).

**Output Parameters**

**Ap**

The upper or lower triangle of \( A \) (as specified by \( \text{uplo} \)) is overwritten by details of the block-diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) (or \( L \)).

**Ipiv**

Array, size at least \( \max(1, n) \). Contains details of the interchanges and the block structure of \( D \). If \( \text{ipiv}[i-1] = k > 0 \), then \( d_{ii} \) is a 1-by-1 block, and the \( i \)-th row and column of \( A \) was interchanged with the \( k \)-th row and column.

If \( \text{uplo} = 'U' \) and \( \text{ipiv}[i] = \text{ipiv}[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \( i \)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

If \( \text{uplo} = 'L' \) and \( \text{ipiv}[i] = \text{ipiv}[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \((i+1)\)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), \( d_{ii} \) is 0. The factorization has been completed, but \( D \) is exactly singular. Division by 0 will occur if you use \( D \) for solving a system of linear equations.

**Application Notes**

The 2-by-2 unit diagonal blocks and the unit diagonal elements of \( U \) and \( L \) are not stored. The remaining elements of \( U \) and \( L \) are stored in the array \( \text{ap} \), but additional row interchanges are required to recover \( U \) or \( L \) explicitly (which is seldom necessary).

If \( \text{ipiv}[i-1] = i \) for all \( i = 1 \ldots n \), then all off-diagonal elements of \( U \) (\( L \)) are stored explicitly in the corresponding elements of the array \( a \).

If \( \text{uplo} = 'U' \), the computed factors \( U \) and \( D \) are the exact factors of a perturbed matrix \( A + E \), where

\[
|E| \leq c(n)\varepsilon|U||D||U^T||P^T|
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

A similar estimate holds for the computed \( L \) and \( D \) when \( \text{uplo} = 'L' \).

The total number of floating-point operations is approximately \((4/3)n^3\).

After calling this routine, you can call the following routines:

- \texttt{?hptrs} to solve \( A^*X = B \)
- \texttt{?hpcon} to estimate the condition number of \( A \)
- \texttt{?hptri} to compute the inverse of \( A \).
See Also
mkl_progress

Matrix Storage Schemes
mkl_?spffrt2, mkl_?spffrtx
Computes the partial LDLᵀ factorization of a symmetric matrix using packed storage.

Syntax
void mkl_sspffrt2 (float *ap , const MKL_INT *n , const MKL_INT *ncolm , float *work , float *work2 );
void mkl_dspffrt2 (double *ap , const MKL_INT *n , const MKL_INT *ncolm , double *work , double *work2 );
void mkl_cspffrt2 (MKL_Complex8 *ap , const MKL_INT *n , const MKL_INT *ncolm , MKL_Complex8 *work , MKL_Complex8 *work2 );
void mkl_zspffrt2 (MKL_Complex16 *ap , const MKL_INT *n , const MKL_INT *ncolm , MKL_Complex16 *work , MKL_Complex16 *work2 );
void mkl_sspffrtx (float *ap , const MKL_INT *n , const MKL_INT *ncolm , float *work , float *work2 );
void mkl_dspffrtx (double *ap , const MKL_INT *n , const MKL_INT *ncolm , double *work , double *work2 );
void mkl_cspffrtx (MKL_Complex8 *ap , const MKL_INT *n , const MKL_INT *ncolm , MKL_Complex8 *work , MKL_Complex8 *work2 );
void mkl_zspffrtx (MKL_Complex16 *ap , const MKL_INT *n , const MKL_INT *ncolm , MKL_Complex16 *work , MKL_Complex16 *work2 );

Include Files
• mkl.h

Description
The routine computes the partial factorization \( A = LDLᵀ \), where \( L \) is a lower triangular matrix and \( D \) is a diagonal matrix.

Caution
The routine assumes that the matrix \( A \) is factorizable. The routine does not perform pivoting and does not handle diagonal elements which are zero, which cause the routine to produce incorrect results without any indication.

Consider the matrix \( A = \begin{pmatrix} a & b \ T \\ b & C \end{pmatrix} \) where \( a \) is the element in the first row and first column of \( A \), \( b \) is a column vector of size \( n - 1 \) containing the elements from the second through \( n \)-th column of \( A \), \( C \) is the lower-right square submatrix of \( A \), and \( I \) is the identity matrix.

The mkl_?spffrt2 routine performs \( n_{colm} \) successive factorizations of the form

\[
A = \begin{pmatrix} a & b \ T \\ b & C \end{pmatrix} = \begin{pmatrix} a_0 & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} a_0^{-1} & 0 \\ 0 & C - ba_0^{-1}b_0 \ T \end{pmatrix} \begin{pmatrix} a & b \ T \\ b & I \end{pmatrix}.
\]
The mkl_?spffrtx routine performs ncolm successive factorizations of the form

\[
A = \begin{pmatrix} a & b^T \\ b & C \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ ba^{-1} & I \end{pmatrix} \begin{pmatrix} a & 0 \\ 0 & C - ba^{-1}b^T \end{pmatrix} \begin{pmatrix} 1 & (ba^{-1})^T \\ 0 & I \end{pmatrix}.
\]

The approximate number of floating point operations performed by real flavors of these routines is

\[
(1/6)ncolm(2ncolm^2 - 6ncolm*n + 3ncolm + 6n^2 - 6n + 7).
\]

The approximate number of floating point operations performed by complex flavors of these routines is

\[
(1/3)ncolm(4ncolm^2 - 12ncolm*n + 9ncolm + 12n^2 - 18n + 8).
\]

**Input Parameters**

- **ap**
  - Array, size at least max(1, n(n+1)/2). The array ap contains the lower triangular part of the matrix A in packed storage (see Matrix Storage Schemes for uplo = 'L').

- **n**
  - The order of matrix A; n \geq 0.

- **ncolm**
  - The number of columns to factor, ncolm \leq n.

- **work, work2**
  - Workspace arrays, size of each at least n.

**Output Parameters**

- **ap**
  - Overwritten by the factor L. The first ncolm diagonal elements of the input matrix A are replaced with the diagonal elements of D. The subdiagonal elements of the first ncolm columns are replaced with the corresponding elements of L. The rest of the input array is updated as indicated in the Description section.

**NOTE**

Specifying ncolm = n results in complete factorization \(A = LDL^T\).

**See Also**

mkl_progress

Matrix Storage Schemes

**Solving Systems of Linear Equations: LAPACK Computational Routines**

This section describes the LAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

**?getrs**

Solves a system of linear equations with an LU-factored square coefficient matrix, with multiple right-hand sides.

**Syntax**

```c
lapack_int LAPACKE_sgetrs (int matrix_layout , char trans , lapack_int n , lapack_int nrhs , const float * a , lapack_int lda , const lapack_int * ipiv , float * b , lapack_int ldb );
```
lapack_int LAPACKE_dgetrs (int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const double * a, lapack_int lda, const lapack_int * ipiv, double * b, lapack_int ldb);

lapack_int LAPACKE_cgetrs (int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zgetrs (int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);

Include Files
• mkl.h

Description
The routine solves for $X$ the following systems of linear equations:

$A*X = B$ if $trans='N'$,

$A^T*X = B$ if $trans='T'$,

$A^H*X = B$ if $trans='C'$ (for complex matrices only).

Before calling this routine, you must call ?getrf to compute the $LU$ factorization of $A$.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major
(LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

trans
Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:
If $trans = 'N'$, then $A*X = B$ is solved for $X$.
If $trans = 'T'$, then $A^T*X = B$ is solved for $X$.
If $trans = 'C'$, then $A^H*X = B$ is solved for $X$.

n
The order of $A$; the number of rows in $B$ ($n \geq 0$).

nrhs
The number of right-hand sides; $nrhs \geq 0$.

a
Array of size max(1, lda*n).
The array $a$ contains $LU$ factorization of matrix $A$ resulting from the call of ?getrf.

b
Array of size max(1,ldb*nrhs) for column major layout, and max(1,lda*n) for row major layout.
The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

lda
The leading dimension of $a$; lda $\geq$ max(1, n).

ldb
The leading dimension of $b$; ldb $\geq$ max(1, n) for column major layout and ldb $\geq$ nrhs for row major layout.
**ipiv**

Array, size at least \( \max(1, n) \). The \( ipiv \) array, as returned by \(?getrf\).

**Output Parameters**

\( b \)

Overwritten by the solution matrix \( X \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

**Application Notes**

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \( (A + E)x = b \), where

\[
|E| \leq c(n)EPA||U|
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{||x - x_0||_\infty}{||x||_\infty} \leq c(n)\text{cond}(A, x)\varepsilon
\]

where \( \text{cond}(A, x) = \frac{||A^{-1}|| \cdot ||A|| \cdot ||x||_\infty}{||x||_\infty \cdot ||A^{-1}||_\infty \cdot ||A||_\infty} = \kappa_\infty(A) \).

Note that \( \text{cond}(A, x) \) can be much smaller than \( \kappa_\infty(A) \); the condition number of \( A^p \) and \( A^H \) might or might not be equal to \( \kappa_\infty(A) \).

The approximate number of floating-point operations for one right-hand side vector \( b \) is \( 2n^2 \) for real flavors and \( 8n^2 \) for complex flavors.

To estimate the condition number \( \kappa_\infty(A) \), call \(?gecon\).

To refine the solution and estimate the error, call \(?gerfs\).

**See Also**

Matrix Storage Schemes

?gbtrs

Solves a system of linear equations with an LU-factored band coefficient matrix, with multiple right-hand sides.

**Syntax**

```c
lapack_int LAPACKE_sgbtrs (int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const float * ab, lapack_int ldab, const lapack_int * ipiv, float * b, lapack_int ldb);

lapack_int LAPACKE_dgbtrs (int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const double * ab, lapack_int ldab, const lapack_int * ipiv, double * b, lapack_int ldb);
```
lapack_int LAPACKE_cgbtrs (int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_float * ab, lapack_int ldab, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zgbtrs (int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_double * ab, lapack_int ldab, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);

Include Files
• mkl.h

Description
The routine solves for $X$ the following systems of linear equations:

$$A*X = B \quad \text{if } \text{trans} = 'N',$$
$$A^T*X = B \quad \text{if } \text{trans} = 'T',$$
$$A^H*X = B \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only).}$$

Here $A$ is an LU-factored general band matrix of order $n$ with $kl$ non-zero subdiagonals and $ku$ nonzero superdiagonals. Before calling this routine, call ?gbtrf to compute the $LU$ factorization of $A$.

Input Parameters
- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **trans**: Must be 'N' or 'T' or 'C'.
- **n**: The order of $A$; the number of rows in $B$; $n \geq 0$.
- **kl**: The number of subdiagonals within the band of $A$; $kl \geq 0$.
- **ku**: The number of superdiagonals within the band of $A$; $ku \geq 0$.
- **nrhs**: The number of right-hand sides; $nrhs \geq 0$.
- **ab**: Array $ab$ size max(1, ldab*n)
The array $ab$ contains elements of the LU factors of the matrix $A$ as returned by gbtrf.
- **b**: Array $b$ size max(1, ldb*nrhs) for column major layout and max(1, ldab*n) for row major layout.
The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.
- **ldab**: The leading dimension of the array $ab$; $ldab \geq 2*kl + ku +1$.
- **ldb**: The leading dimension of $b$; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
- **ipiv**: Array, size at least max(1, n). The $ipiv$ array, as returned by ?gbtrf.

Output Parameters
- **b**: Overwritten by the solution matrix $X$. 462
Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, parameter i had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations (A + E)x = b, where

\[ |E| \leq c(k_l + k_u + 1)\varepsilon\|L\|\|U\| \]

c(k) is a modest linear function of k, and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[ \frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(k_l + k_u + 1)\text{cond}(A, x)\varepsilon \]

where \( \text{cond}(A, x) = \frac{\|A^{-1}\|\|A\|}{\|x\|_\infty / \|Ax\|_\infty \leq \|A^{-1}\|\|A\|_\infty} = \kappa_x(A) \).

Note that \( \text{cond}(A, x) \) can be much smaller than \( \kappa_x(A) \); the condition number of \( A^T \) and \( A^H \) might or might not be equal to \( \kappa_x(A) \).

The approximate number of floating-point operations for one right-hand side vector is 2n(ku + 2kl) for real flavors. The number of operations for complex flavors is 4 times greater. All these estimates assume that kl and ku are much less than \( \min(m, n) \).

To estimate the condition number \( \kappa_x(A) \), call ?gbcon.

To refine the solution and estimate the error, call ?gbrfs.

See Also

Matrix Storage Schemes

?gttrs

Solves a system of linear equations with a tridiagonal coefficient matrix using the LU factorization computed by ?gttrf.

Syntax

```c
lapack_int LAPACKE_sgttrs (int matrix_layout , char trans , lapack_int n , lapack_int nrhs , const float * dl , const float * d , const float * du , const float * du2 , const lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dgttrs (int matrix_layout , char trans , lapack_int n , lapack_int nrhs , const double * dl , const double * d , const double * du , const double * du2 , const lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_cgttrs (int matrix_layout , char trans , lapack_int n , lapack_int nrhs , const lapack_complex_float * d1 , const lapack_complex_float * d , const lapack_complex_float * du , const lapack_complex_float * du2 , const lapack_int * ipiv , lapack_complex_float * b , lapack_int ldb );
```
lapack_int LAPACKE_zgttrs (int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double * dl, const lapack_complex_double * d, const lapack_complex_double * du, const lapack_complex_double * du2, const lapack_int * ipiv, lapack_complex_double * b, lapack_int LDB);

**Include Files**
- mkl.h

**Description**

The routine solves for $X$ the following systems of linear equations with multiple right hand sides:

- $A*X = B$ if $trans='N'$,
- $A^T*X = B$ if $trans='T'$,
- $A^H*X = B$ if $trans='C'$ (for complex matrices only).

Before calling this routine, you must call **gttrf** to compute the $LU$ factorization of $A$.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout for array $b$ is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **trans**
  Must be 'N' or 'T' or 'C'.
  Indicates the form of the equations:
  - If $trans='N'$, then $A*X = B$ is solved for $X$.
  - If $trans='T'$, then $A^T*X = B$ is solved for $X$.
  - If $trans='C'$, then $A^H*X = B$ is solved for $X$.

- **n**
  The order of $A$; $n \geq 0$.

- **nrhs**
  The number of right-hand sides, that is, the number of columns in $B$; $nrhs \geq 0$.

- **dl, d, du, du2**
  Arrays: $dl(n-1), d(n), du(n-1), du2(n-2)$.
  The array $dl$ contains the $(n - 1)$ multipliers that define the matrix $L$ from the $LU$ factorization of $A$.
  The array $d$ contains the $n$ diagonal elements of the upper triangular matrix $U$ from the $LU$ factorization of $A$.
  The array $du$ contains the $(n - 1)$ elements of the first superdiagonal of $U$.
  The array $du2$ contains the $(n - 2)$ elements of the second superdiagonal of $U$.

- **b**
  Array of size max(1, $LDB*nrhs$) for column major layout and max(1, $n*LDB$) for row major layout. Contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

- **LDB**
  The leading dimension of $b$: $LDB \geq \max(1, n)$ for column major layout and $LDB \geq nrhs$ for row major layout.
**ipiv**

Array, size \((n)\). The \(ipiv\) array, as returned by \(?gttrf\).

**Output Parameters**

\(b\)  
Overwritten by the solution matrix \(X\).

**Return Values**

This function returns a value \(info\).

If \(info=0\), the execution is successful.

If \(info = -i\), parameter \(i\) had an illegal value.

**Application Notes**

For each right-hand side \(b\), the computed solution is the exact solution of a perturbed system of equations \((A + E)x = b\), where

\[
|E| \leq c(n)\varepsilon|L||U|
\]

\(c(n)\) is a modest linear function of \(n\), and \(\varepsilon\) is the machine precision.

If \(x_0\) is the true solution, the computed solution \(x\) satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(kL + ku + 1) \text{cond} (A, x)\varepsilon
\]

where \(\text{cond}(A, x) = \|A^{-1}\| |A| |x| / \|x\|_\infty\| |A^{-1}|\|_\infty \|A\|_\infty = \kappa_x(A)\).

Note that \(\text{cond}(A, x)\) can be much smaller than \(\kappa_x(A)\); the condition number of \(A^T\) and \(A^H\) might or might not be equal to \(\kappa_x(A)\).

The approximate number of floating-point operations for one right-hand side vector \(b\) is \(7n\) (including \(n\) divisions) for real flavors and \(34n\) (including \(2n\) divisions) for complex flavors.

To estimate the condition number \(\kappa_x(A)\), call \(?gtcon\).

To refine the solution and estimate the error, call \(?gtrfs\).

**See Also**

Matrix Storage Schemes

\(?dttrsbr\)

Solves a system of linear equations with a diagonally dominant tridiagonal coefficient matrix using the LU factorization computed by \(?dttrfb\).

**Syntax**

```c
void sdttrsb (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const float * dl, const float * d, const float * du, float * b, const MKL_INT * ldb, MKL_INT * info);
void ddttrsb (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const double * dl, const double * d, const double * du, double * b, const MKL_INT * ldb, MKL_INT * info);
```
void cdttrsb (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const MKL_Complex8 * dl, const MKL_Complex8 * d, const MKL_Complex8 * du, MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * info);
void zdttrsb (const char * trans, const MKL_INT * n, const MKL_INT * nrhs, const MKL_Complex16 * dl, const MKL_Complex16 * d, const MKL_Complex16 * du, MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT * info );

Include Files
• mkl.h

Description
The ?dttrsb routine solves the following systems of linear equations with multiple right hand sides for X:

\[ A \times X = B \quad \text{if } \text{trans} = 'N', \]
\[ A^T \times X = B \quad \text{if } \text{trans} = 'T', \]
\[ A^H \times X = B \quad \text{if } \text{trans} = 'C' \text{ (for complex matrices only)}. \]

Before calling this routine, call ?dttrfb to compute the factorization of A.

Input Parameters

trans
Must be 'N' or 'T' or 'C'.
Indicates the form of the equations solved for X:
If trans = 'N', then \( A \times X = B \).
If trans = 'T', then \( A^T \times X = B \).
If trans = 'C', then \( A^H \times X = B \).

n
The order of A; \( n \geq 0 \).
nrhs
The number of right-hand sides, that is, the number of columns in B; \( nrhs \geq 0 \).

dl, d, du
Arrays: \( dl(n-1), d(n), du(n-1) \).
The array dl contains the \((n-1)\) multipliers that define the matrices \( L_1, L_2 \) from the factorization of A.
The array d contains the \( n \) diagonal elements of the upper triangular matrix \( U \) from the factorization of A.
The array du contains the \((n-1)\) elements of the superdiagonal of \( U \).

b
Array of size max(1, ldb*nrhs). Contains the matrix B whose columns are the right-hand sides for the systems of equations.

ldb
The leading dimension of b; \( ldb \geq \max(1, n) \).

Output Parameters

b
Overwritten by the solution matrix X.
info
If info = 0, the execution is successful.
If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**?potrs**
Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

**Syntax**

```c
lapack_int LAPACKE_spotrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * a, lapack_int lda, float * b, lapack_int ldb );
```

```c
lapack_int LAPACKE_dpotrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * a, lapack_int lda, double * b, lapack_int ldb );
```

```c
lapack_int LAPACKE_cpotrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb );
```

```c
lapack_int LAPACKE_zpotrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb );
```

**Include Files**
- mkl.h

**Description**
The routine solves for \( X \) the system of linear equations \( A \cdot X = B \) with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix \( A \), given the Cholesky factorization of \( A \):

\[
A = U^T \cdot U \quad \text{for real data,} \quad A = U^H \cdot U \quad \text{for complex data}
\]

\[
A = L \cdot L^T \quad \text{for real data,} \quad A = L \cdot L^H \quad \text{for complex data}
\]

if \( \text{uplo} = \text{'U'} \)

where \( L \) is a lower triangular matrix and \( U \) is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix \( B \).

Before calling this routine, you must call ?potrf to compute the Cholesky factorization of \( A \).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **uplo**
  Must be \( \text{'U'} \) or \( \text{'L'} \).
  Indicates how the input matrix \( A \) has been factored:
  - If \( \text{uplo} = \text{'U'} \), \( U \) is stored, where \( A = U^T \cdot U \) for real data, \( A = U^H \cdot U \) for complex data.
  - If \( \text{uplo} = \text{'L'} \), \( L \) is stored, where \( A = L \cdot L^T \) for real data, \( A = L \cdot L^H \) for complex data.

- **n**
  The order of matrix \( A; n \geq 0 \).

- **nrhs**
  The number of right-hand sides (\( nrhs \geq 0 \)).

- **a**
  Array \( A \) of size at least \( \max(1, lda \cdot n) \)
The array $a$ contains the factor $U$ or $L$ (see uplo) as returned by potrf.

$lda$

The leading dimension of $a$. $lda \geq \max(1, n)$.

$b$

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations. The size of $b$ must be at least $\max(1, ldb \times nrhs)$ for column major layout and $\max(1, ldb \times n)$ for row major layout.

$ldb$

The leading dimension of $b$. $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

**Output Parameters**

$b$

Overwritten by the solution matrix $X$.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

**Application Notes**

If uplo = 'U', the computed solution for each right-hand side $b$ is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\epsilon |UH||U|$$

$c(n)$ is a modest linear function of $n$, and $\epsilon$ is the machine precision.

A similar estimate holds for uplo = 'L'. If $x_0$ is the true solution, the computed solution $x$ satisfies this error bound:

$$\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) cond(A, x)\epsilon$$

where $cond(A, x) = \frac{\|A^{-1}\| |A| |x|}{\| |x| \|_\infty / | \|A^{-1}\|_\infty | |A| \|_\infty = \kappa_\infty(A)$.

Note that $cond(A, x)$ can be much smaller than $\kappa_\infty(A)$. The approximate number of floating-point operations for one right-hand side vector $b$ is $2n^2$ for real flavors and $8n^2$ for complex flavors.

To estimate the condition number $\kappa_\infty(A)$, call ?pocon.

To refine the solution and estimate the error, call ?porfs.

**See Also**

Matrix Storage Schemes

?pftrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix using the Rectangular Full Packed (RFP) format.
Syntax

```c
lapack_int LAPACKE_spftrs (int matrix_layout, char transr, char uplo, lapack_int n,
 lapack_int nrhs, const float * a, float * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_dpftrs (int matrix_layout, char transr, char uplo, lapack_int n,
 lapack_int nrhs, const double * a, double * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_cpftrs (int matrix_layout, char transr, char uplo, lapack_int n,
 lapack_int nrhs, const lapack_complex_float * a, lapack_complex_float * b,
 lapack_int ldb);
```

```c
lapack_int LAPACKE_zpftrs (int matrix_layout, char transr, char uplo, lapack_int n,
 lapack_int nrhs, const lapack_complex_double * a, lapack_complex_double * b,
 lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine solves a system of linear equations $A X = B$ with a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix $A$ using the Cholesky factorization of $A$:

- $A = U^T U$ for real data, $A = U^H U$ for complex data if `uplo='U'`
- $A = L L^T$ for real data, $A = L L^H$ for complex data if `uplo='L'`

Before calling ?pftrs, you must call ?pftrf to compute the Cholesky factorization of $A$. $L$ stands for a lower triangular matrix and $U$ for an upper triangular matrix.

The matrix $A$ is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- `transr` Must be 'N', 'T' (for real data) or 'C' (for complex data).

  - If `transr = 'N'`, the untransposed factor of $A$ is stored in RFP format.
  - If `transr = 'T'`, the transposed factor of $A$ is stored in RFP format.
  - If `transr = 'C'`, the conjugate-transposed factor of $A$ is stored in RFP format.

- `uplo` Must be 'U' or 'L'.

  Indicates how the input matrix $A$ has been factored:

  - If `uplo = 'U'`, $U$ is stored, where $A = U^T U$ for real data, $A = U^H U$ for complex data.
  - If `uplo = 'L'`, $L$ is stored, where $A = L L^T$ for real data, $A = L L^H$ for complex data.

- `n` The order of the matrix $A$; $n \geq 0$. 


nrhs

The number of right-hand sides, that is, the number of columns of the matrix $B$; $nrhs \geq 0$.

$a$

Array $a$ of size $\max(1, n(n + 1)/2)$.
The array $a$ contains, in the RFP format, the factor $U$ or $L$ obtained by factorization of matrix $A$.

$b$

The array $b$ of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$ldb$

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

**Output Parameters**

$b$

The solution matrix $X$.

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

**See Also**

Matrix Storage Schemes

?pptrs

Solves a system of linear equations with a packed Cholesky-factored symmetric (Hermitian) positive-definite coefficient matrix.

**Syntax**

```c
lapack_int LAPACKE_sptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * ap, float * b, lapack_int ldb);
lapack_int LAPACKE_dptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * ap, double * b, lapack_int ldb);
lapack_int LAPACKE_cpptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * ap, lapack_complex_float * b, lapack_int ldb);
lapack_int LAPACKE_zpptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * ap, lapack_complex_double * b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine solves for $X$ the system of linear equations $A*X = B$ with a packed symmetric positive-definite or, for complex data, Hermitian positive-definite matrix $A$, given the Cholesky factorization of $A$:

$A = U^T*U$ for real data, $A = U^H*U$ for complex data  \hspace{1cm}$\text{if } uplo='U'$

$A = L*L^T$ for real data, $A = L*L^H$ for complex data  \hspace{1cm}$\text{if } uplo='L'$
where $L$ is a lower triangular matrix and $U$ is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix $B$.

Before calling this routine, you must call ?pptrf to compute the Cholesky factorization of $A$.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**: Must be 'U' or 'L'. Indicates how the input matrix $A$ has been factored:
  - If $uplo = 'U'$, $U$ is stored, where $A = U^T * U$ for real data, $A = U^H * U$ for complex data.
  - If $uplo = 'L'$, $L$ is stored, where $A = L * L^T$ for real data, $A = L * L^H$ for complex data.

- **n**: The order of matrix $A$; $n \geq 0$.

- **nrhs**: The number of right-hand sides ($nrhs \geq 0$).

- **ap, b**: The size of $ap$ must be at least $\max(1, n(n+1)/2)$.
  The array $ap$ contains the factor $U$ or $L$, as specified by $uplo$, in packed storage (see Matrix Storage Schemes).

- **b**: The array $b$ of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

- **ldb**: The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

**Output Parameters**

- **b**: Overwritten by the solution matrix $X$.

**Return Values**

This function returns a value $info$.

- If $info = 0$, the execution is successful.
- If $info = -i$, parameter $i$ had an illegal value.

**Application Notes**

If $uplo = 'U'$, the computed solution for each right-hand side $b$ is the exact solution of a perturbed system of equations $(A + E)x = b$, where

$$|E| \leq c(n)\varepsilon |U^T||U|$$

$c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision.

A similar estimate holds for $uplo = 'L'$.

If $x_0$ is the true solution, the computed solution $x$ satisfies this error bound:
\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond} (A, x)\varepsilon
\]

where \(\text{cond} (A, x) = \frac{\|A^{-1}\| \|x\| \|x\| / \|Ax\| \|A\|}{\|A^{-1}\| \|A\|} = \kappa_x(A)\).

Note that \(\text{cond} (A, x)\) can be much smaller than \(\kappa_x(A)\).

The approximate number of floating-point operations for one right-hand side vector \(b\) is \(2n^2\) for real flavors and \(8n^2\) for complex flavors.

To estimate the condition number \(\kappa_x(A)\), call \?ppcon.

To refine the solution and estimate the error, call \?pprfs.

See Also

Matrix Storage Schemes

\?pbtrs

Solves a system of linear equations with a Cholesky-factored symmetric (Hermitian) positive-definite band coefficient matrix.

Syntax

```c
lapack_int LAPACKE_spbtrs (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, const float * ab, lapack_int ldab, float * b, lapack_int ldb );

lapack_int LAPACKE_dpbtrs (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, const double * ab, lapack_int ldab, double * b, lapack_int ldb );

lapack_int LAPACKE_cpbtrs (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_float * ab, lapack_int ldab, lapack_complex_float * b, lapack_int ldb );

lapack_int LAPACKE_zpbtrs (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_double * ab, lapack_int ldab, lapack_complex_double * b, lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for real data a system of linear equations \(A^*X = B\) with a symmetric positive-definite or, for complex data, Hermitian positive-definite band matrix \(A\), given the Cholesky factorization of \(A\):

\[
A = U^T \ast U \text{ for real data, } A = U^H \ast U \text{ for complex data}
\]

\[
A = L \ast L^T \text{ for real data, } A = L \ast L^H \text{ for complex data}
\]

where \(L\) is a lower triangular matrix and \(U\) is upper triangular. The system is solved with multiple right-hand sides stored in the columns of the matrix \(B\).

Before calling this routine, you must call \?pbtrf to compute the Cholesky factorization of \(A\) in the band storage form.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**uplo**
Must be 'U' or 'L'.
Indicates how the input matrix A has been factored:
If uplo = 'U', U is stored in ab, where \( A = U^T \cdot U \) for real matrices and \( A = U^H \cdot U \) for complex matrices.
If uplo = 'L', L is stored in ab, where \( A = L \cdot L^T \) for real matrices and \( A = L \cdot L^H \) for complex matrices.

**n**
The order of matrix A; \( n \geq 0 \).

**kd**
The number of superdiagonals or subdiagonals in the matrix A; \( kd \geq 0 \).

**nrhs**
The number of right-hand sides; \( nrhs \geq 0 \).

**ab**
Array ab is of size max (1, ldab*n).
The array ab contains the Cholesky factor, as returned by the factorization routine, in band storage form.
The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

**b**
The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.
The size of b is at least max(1, ldb*nrhs) for column major layout and max(1, ldb*n) for row major layout.

**ldab**
The leading dimension of the array ab; \( ldab \geq kd + 1 \).

**ldb**
The leading dimension of b; \( ldb \geq max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Output Parameters

**b**
Overwritten by the solution matrix X.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, parameter i had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations \((A + E)x = b\), where

\[
|E| \leq c(kd + 1)\varepsilon P|U^H||U| \quad \text{or} \quad |E| \leq c(kd + 1)\varepsilon P|L^T||L|
\]

\(c(k)\) is a modest linear function of \(k\), and \(\varepsilon\) is the machine precision.
If \(x_0\) is the true solution, the computed solution \(x\) satisfies this error bound:
\[ \frac{\| x - x_0 \|_\infty}{\| x \|_\infty} \leq c(kd + 1) \text{cond} (A, x)\varepsilon \]

where \( \text{cond} (A, x) = || A^{-1}|| A \| x \|_\infty / ||x||_\infty \leq || A^{-1}||_\infty \| A \| \_\infty = \kappa_\infty (A) \).

Note that \( \text{cond} (A, x) \) can be much smaller than \( \kappa_\infty (A) \).

The approximate number of floating-point operations for one right-hand side vector is \( 4n*kd \) for real flavors and \( 16n*kd \) for complex flavors.

To estimate the condition number \( \kappa_\infty (A) \), call \?pbcon.

To refine the solution and estimate the error, call \?pbrfs.

See Also
Matrix Storage Schemes

\?pttrs
Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix using the factorization computed by \?pttrf.

Syntax

\[
\begin{align*}
\text{lapack_int LAPACKE_spttrs} & ( \text{int matrix_layout, lapack_int n, lapack_int nrhs, const float* d, const float* e, float* b, lapack_int ldb } ); \\
\text{lapack_int LAPACKE_dpttrs} & ( \text{int matrix_layout, lapack_int n, lapack_int nrhs, const double* d, const double* e, double* b, lapack_int ldb } ); \\
\text{lapack_int LAPACKE_cpttrs} & ( \text{int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float* d, const lapack_complex_float* e, lapack_complex_float* b, lapack_int ldb } ); \\
\text{lapack_int LAPACKE_zpttrs} & ( \text{int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double* d, const lapack_complex_double* e, lapack_complex_double* b, lapack_int ldb } );
\end{align*}
\]

Include Files

- mkl.h

Description

The routine solves for \( X \) a system of linear equations \( A^*X = B \) with a symmetric (Hermitian) positive-definite tridiagonal matrix \( A \). Before calling this routine, call \?pttrf to compute the \( L*D*L^T \) or \( U^*D*U \) for real data and the \( L*D*L^H \) or \( U^*D*U \) factorization of \( A \) for complex data.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Used for cptrs/zptrs only. Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix \( A \) is stored and how \( A \) is factored:
If uplo = 'U', the array e stores the conjugated values of the superdiagonal of U, and A is factored as $U^H \cdot D \cdot U$.

If uplo = 'L', the array e stores the subdiagonal of L, and A is factored as $L \cdot D \cdot L^H$.

$n$  
The order of A; $n \geq 0$.

$nrhs$  
The number of right-hand sides, that is, the number of columns of the matrix B; $nrhs \geq 0$.

d  
Array, dimension ($n$). Contains the diagonal elements of the diagonal matrix D from the factorization computed by ?pttrf.

e  
Array e is of size ($n - 1$).

The array e contains the $(n - 1)$ sub-diagonal elements of the unit bidiagonal factor L or the conjugated values of the superdiagonal of U from the factorization computed by ?pttrf (see uplo).

e, b  
The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

The size of b is at least max(1, $ldb \cdot nrhs$) for column major layout and max(1, $ldb \cdot n$) for row major layout.

$ldb$  
The leading dimension of b; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

b  
Overwritten by the solution matrix X.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, parameter i had an illegal value.

See Also

Matrix Storage Schemes

?sytrs
Solves a system of linear equations with a $UDU^T$- or $LDL^T$-factored symmetric coefficient matrix.

Syntax

lapack_int LAPACKE_ssytrs (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , const float * a , lapack_int lda , const lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dsytrs (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , const double * a , lapack_int lda , const lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_csytrs (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , const lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv , lapack_complex_float * b , lapack_int ldb );

LAPACK Routines 3
include mkl.h

Description

The routine solves for X the system of linear equations \( A*X = B \) with a symmetric matrix \( A \), given the Bunch-Kaufman factorization of \( A \):

if \( uplo = 'U' \), \( A = U*D*U^T \)

if \( uplo = 'L' \), \( A = L*D*L^T \)

where \( U \) and \( L \) are upper and lower triangular matrices with unit diagonal and \( D \) is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix \( B \).

You must supply to this routine the factor \( U \) (or \( L \)) and the array \( ipiv \) returned by the factorization routine \?sytrf.

Input Parameters

- \texttt{matrix\_layout}\n  Specifies whether matrix storage layout is row major \( (\text{LAPACK\_ROW\_MAJOR}) \) or column major \( (\text{LAPACK\_COL\_MAJOR}) \).

- \texttt{uplo}\n  Must be \( 'U' \) or \( 'L' \).

- \texttt{n}\n  The order of matrix \( A \); \( n \geq 0 \).

- \texttt{nrhs}\n  The number of right-hand sides; \( nrhs \geq 0 \).

- \texttt{ipiv}\n  Array, size at least \( \max(1, n) \). The \( ipiv \) array, as returned by \?sytrf.

- \texttt{a}\n  The array \( a \) of size \( \max(1, lda*n) \) contains the factor \( U \) or \( L \) (see \texttt{uplo})

- \texttt{b}\n  The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the system of equations.

- \texttt{lda}\n  The leading dimension of \( a \); \( lda \geq \max(1, n) \).

- \texttt{ldb}\n  The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Output Parameters

- \texttt{b}\n  Overwritten by the solution matrix \( X \).
**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

**Application Notes**

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \( (A + E)x = b \), where

\[
|E| \leq c(n) \|P\| |U|^T |P^T \text{ or } |E| \leq c(n) \|P\| |L| |D| |U|^T |P^T
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{\| x - x_0 \|_\infty}{\| x \|_\infty} \leq c(n) \text{cond} (A, x) \varepsilon
\]

where \( \text{cond}(A, x) = \| A^{-1} \| \|A\| \| x \|_\infty / \| x \|_\infty \leq \| A^{-1} \| \|A\|_\infty = \kappa_\infty (A) \).

Note that \( \text{cond}(A, x) \) can be much smaller than \( \kappa_\infty (A) \).

The total number of floating-point operations for one right-hand side vector is approximately \( 2n^2 \) for real flavors or \( 8n^2 \) for complex flavors.

To estimate the condition number \( \kappa_\infty (A) \), call \(?sycon\).

To refine the solution and estimate the error, call \(?syrfs\).

**See Also**

Matrix Storage Schemes

\( ?\text{sytrs}\_\text{aa} \)

Solves a system of linear equations \( A * X = B \) with a symmetric matrix.

```c
lapack_int LAPACKE_ssytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * A, lapack_int lda, const lapack_int * ipiv, float * B, lapack_int ldb);

lapack_int LAPACKE_dsytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * A, lapack_int lda, const lapack_int * ipiv, double * B, lapack_int ldb);

lapack_int LAPACKE_csytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);

lapack_int LAPACKE_zsytrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

**Description**

\( ?\text{sytrs}\_\text{aa} \) solves a system of linear equations \( A * X = B \) with a symmetric matrix \( A \) using the factorization \( A = U^T U^T \) or \( A = L^T L^T \) computed by \(?\text{sytrf}\_\text{aa}\).
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**uplo**
Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.
- = 'U': Upper triangular; the form is \( A = U^T \cdot U \).
- = 'L': Lower triangular; the form is \( A = L^T \cdot L \).

**n**
The order of the matrix \( A \). \( n \geq 0 \).

**nrhs**
The number of right-hand sides; that is, the number of columns of the matrix \( B \). \( nrhs \geq 0 \).

**A**
Array of size \( \max(1, lda \cdot n) \). Details of factors computed by ?sytrf_aa.

**lda**
The leading dimension of the array \( A \).

**ipiv**
Array of size \( n \). Details of the interchanges as computed by ?sytrf_aa.

**B**
Array of size \( \max(1, ldb \cdot nrhs) \). On entry, the right-hand side matrix \( B \).

**ldb**
The leading dimension of the array \( B \). \( ldb \geq \max(1, n) \) for column-major layout and \( ldb \geq nrhs \) for row-major layout.

Output Parameters

**B**
On exit, the solution matrix \( X \).

Return Values
This function returns a value \( info \).
- = 0: Successful exit.
- < 0: If \( info = -i \), the \( i \)th argument had an illegal value.

**?sytrs_rook**
Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax

```c
lapack_int LAPACKE_ssytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * a, lapack_int lda, const lapack_int * ipiv, float * b, lapack_int ldb);
lapack_int LAPACKE_dsytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * a, lapack_int lda, const lapack_int * ipiv, double * b, lapack_int ldb);
lapack_int LAPACKE_csytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);
lapack_int LAPACKE_zsytrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);
```
Include Files

- mkl.h

Description

The routine solves a system of linear equations \( A \times X = B \) with a symmetric matrix \( A \), using the factorization \( A = U \times D \times U^T \) or \( A = L \times D \times L^T \) computed by \?sytrf_rook. \n
Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout for array \( b \) is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be ‘U’ or ‘L’.
  Indicates how the input matrix \( A \) has been factored:
  - If \( uplo = 'U' \), the factorization is of the form \( A = U \times D \times U^T \).
  - If \( uplo = 'L' \), the factorization is of the form \( A = L \times D \times L^T \).

- **n**
  The order of matrix \( A \); \( n \geq 0 \).

- **nrhs**
  The number of right-hand sides; \( nrhs \geq 0 \).

- **ipiv**
  Array, size at least \( \max(1, n) \). The \( ipiv \) array, as returned by \?sytrf_rook.

- **a, b**
  Arrays: \( a \), size \( (lda \times n) \), \( b \) size \( (ldb \times nrhs) \).
  The array \( a \) contains the block diagonal matrix \( D \) and the multipliers used to obtain \( U \) or \( L \) as computed by \?sytrf_rook (see \( uplo \)).
  The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the system of equations.

- **lda**
  The leading dimension of \( a \); \( lda \geq \max(1, n) \).

- **ldb**
  The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Output Parameters

- **b**
  Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).
If \( info=0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The total number of floating-point operations for one right-hand side vector is approximately \( 2n^2 \) for real flavors or \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes
?hetrs
Solves a system of linear equations with a UDU<sup>T</sup>- or LDL<sup>T</sup>-factored Hermitian coefficient matrix.

**Syntax**

```c
lapack_int LAPACKE_chetrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zhetrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine solves for \( X \) the system of linear equations \( A \cdot X = B \) with a Hermitian matrix \( A \), given the Bunch-Kaufman factorization of \( A \):

\[
\begin{align*}
\text{if } \text{uplo}= 'U', & \quad A = U \cdot D \cdot U^H \\
\text{if } \text{uplo}= 'L', & \quad A = L \cdot D \cdot L^H,
\end{align*}
\]

where \( U \) and \( L \) are upper and lower triangular matrices with unit diagonal and \( D \) is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix \( B \). You must supply to this routine the factor \( U \) (or \( L \)) and the array \( ipiv \) returned by the factorization routine ?hetrf.

**Input Parameters**

- `matrix_layout`: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo`: Must be 'U' or 'L'. Indicates how the input matrix \( A \) has been factored:
  - If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular factor \( U \) of the factorization \( A = U \cdot D \cdot U^H \).
  - If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular factor \( L \) of the factorization \( A = L \cdot D \cdot L^H \).
- `n`: The order of matrix \( A \); \( n \geq 0 \).
- `nrhs`: The number of right-hand sides; \( nrhs \geq 0 \).
- `ipiv`: Array, size at least \( \max(1, n) \). The \( ipiv \) array, as returned by ?hetrf.
- `a`: The array \( a \) of size \( \max(1, \text{lda} \times n) \) contains the factor \( U \) or \( L \) (see \( uplo \)).
- `b`: The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the system of equations.
The size of $b$ is at least $\max(1, \text{ldb} \times \text{nrhs})$ for column major layout and $\max(1, \text{ldb} \times n)$ for row major layout.

$\text{lda}$

The leading dimension of $a$; $\text{lda} \geq \max(1, n)$.

$\text{ldb}$

The leading dimension of $b$; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.

**Output Parameters**

$b$

Overwritten by the solution matrix $X$.

**Return Values**

This function returns a value $\text{info}$.

If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, parameter $i$ had an illegal value.

**Application Notes**

For each right-hand side $b$, the computed solution is the exact solution of a perturbed system of equations $(A + E)x = b$, where

\[ |E| \leq c(n)\varepsilon|U| |D| |P^T| \text{ or } |E| \leq c(n)\varepsilon|L| |D| |L^H| P^T \]

$c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision.

If $x_0$ is the true solution, the computed solution $x$ satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x)\varepsilon
\]

where $\text{cond}(A, x) = \| |A^{-1}| |A| |x| \| / \| |x| \| \leq \| |A^{-1}| |A| \| = \kappa_\infty(A)$.

Note that $\text{cond}(A, x)$ can be much smaller than $\kappa_\infty(A)$.

The total number of floating-point operations for one right-hand side vector is approximately $8n^2$.

To estimate the condition number $\kappa_\infty(A)$, call `?hecon`.

To refine the solution and estimate the error, call `?herfs`.

**See Also**

**Matrix Storage Schemes**

`?hetrs_aa`

BSolves a system of linear equations $A^*X = b$ with a complex Hermitian matrix.

LAPACK Declares

```c
LAPACK_DECL lapack_int LAPACKE_chetrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb );
```

LAPACK Declares

```c
LAPACK_DECL lapack_int LAPACKE_zhetrs_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb );
```
Description

?hetrs_aa solves a system of linear equations $A^*X = X$ with a complex Hermitian matrix $A$ using the factorization $A = U * T * U^H$ or $A = L * T * L^H$ computed by ?hetrf_aa.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.

If $uplo = 'U'$: Upper triangular of the form $A = U * T * U^H$.

If $uplo = 'L'$: Lower triangular of the form $A = L * T * L^H$.

n

The order of the matrix $A$. $n \geq 0$.

nrhs

The number of right hand sides: the number of columns of the matrix $b$. $nrhs \geq 0$.

a

Array of size $lda*n$. Details of factors computed by ?hetrf_aa.

lda

The leading dimension of the array $a$. $lda \geq \max(1, n)$.

ipiv

Array of size $(n)$. Details of the interchanges as computed by ?hetrf_aa.

b

Array of size $ldb*nrhs$. On entry, the right hand side matrix $B$.

ldb

The leading dimension of the array $b$. $ldb \geq \max(1, n)$.

work

See Syntax - Workspace. Array of size $(\max(1, lwork))$.

lwork

See Syntax - Workspace. $lwork \geq \max(1, 3*n-2)$.

Output Parameters

b

On exit, the solution matrix $X$.

Return Values

This function returns a value $info$.

If $info = 0$: successful exit.

If $info < 0$: if $info = -i$, the $i$-th argument had an illegal value.

Syntax - Workspace

Use this interface if you want to explicitly provide the workspace array.

LAPACK DECL lapack_int LAPACKE_chetrs_aa_work (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * work, lapack_int lwork );

LAPACK DECL lapack_int LAPACKE_zhetrs_aa_work (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb, lapack_complex_double * work, lapack_int lwork );
**?hetrs_rook**

Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix.

**Syntax**

```c
lapack_int LAPACKE_chetrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zhetrs_rook (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine solves for a system of linear equations \( A \cdot X = B \) with a complex Hermitian matrix \( A \) using the factorization \( A = U \cdot D \cdot U^H \) or \( A = L \cdot D \cdot L^H \) computed by \( ?hetrf_rook \).

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout for array \( b \) is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo` Must be \('U'\) or \('L'\).
  - Indicates how the input matrix \( A \) has been factored:
    - If \( uplo = 'U' \), the factorization is of the form \( A = U \cdot D \cdot U^H \).
    - If \( uplo = 'L' \), the factorization is of the form \( A = L \cdot D \cdot L^H \).
- `n` The order of matrix \( A; n \geq 0 \).
- `nrhs` The number of right-hand sides; \( nrhs \geq 0 \).
- `ipiv` Array, size at least \( \max(1, n) \).
  - The `ipiv` array, as returned by \( ?hetrf_rook \).
- `a`, `b` Arrays: \( (lda \cdot n) \), \( (ldb \cdot nrhs) \).
  - The array \( a \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as computed by \( ?hetrf_rook \) (see `uplo`).
  - The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the system of equations.
- `lda` The leading dimension of \( a; lda \geq \max(1, n) \).
- `ldb` The leading dimension of \( b; ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

**Output Parameters**

- `b` Overwritten by the solution matrix \( X \).
Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

?sytrs2
Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix.

Syntax
lapack_int LAPACKE_ssytrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * a, lapack_int lda, const lapack_int * ipiv, float * b, lapack_int ldb);

lapack_int LAPACKE_dsytrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * a, lapack_int lda, const lapack_int * ipiv, double * b, lapack_int ldb);

lapack_int LAPACKE_csytrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zsytrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);

Include Files
- mkl.h

Description
The routine solves a system of linear equations $A*X = B$ with a symmetric matrix $A$ using the factorization of $A$:

if uplo='U', \[ A = U*D*U^T \]
if uplo='L', \[ A = L*D*L^T \]

where
- $U$ and $L$ are upper and lower triangular matrices with unit diagonal
- $D$ is a symmetric block-diagonal matrix.

The factorization is computed by ?sytrf.

Input Parameters

$matrix_layout$
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

$uplo$
Must be 'U' or 'L'.
Indicates how the input matrix $A$ has been factored:
If uplo = 'U', the array $a$ stores the upper triangular factor $U$ of the factorization $A = U*D*U^T$. 
If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular factor \( L \) of the factorization \( A = L*D*L^T \).

\( n \)

The order of matrix \( A \); \( n \geq 0 \).

\( \text{nrhs} \)

The number of right-hand sides; \( \text{nrhs} \geq 0 \).

\( a \)

The array \( a \) of size \( \max(1, \text{lda} \times n) \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as computed by \(?sytrf\).

\( b \)

The array \( b \) contains the right-hand side matrix \( B \).

The size of \( b \) is at least \( \max(1, \text{ldb} \times \text{nrhs}) \) for column major layout and \( \max(1, \text{ldb} \times n) \) for row major layout.

\text{lda}

The leading dimension of \( a \); \( \text{lda} \geq \max(1, n) \).

\text{ldb}

The leading dimension of \( b \); \( \text{ldb} \geq \max(1, n) \) for column major layout and \( \text{ldb} \geq \text{nrhs} \) for row major layout.

\text{ipiv}

Array of size \( n \). The \text{ipiv} array contains details of the interchanges and the block structure of \( D \) as determined by \(?sytrf\).

**Output Parameters**

\( b \)

Overwritten by the solution matrix \( X \).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

**See Also**

\(?sytrf\)

**Matrix Storage Schemes**

\(?hetrs2\)

_Solves a system of linear equations with a UDU- or LDL-factored Hermitian coefficient matrix._

**Syntax**

```c
lapack_int LAPACKE_chetrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb );

lapack_int LAPACKE_zhetrs2 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb );
```

**Include Files**

- mkl.h
Description

The routine solves a system of linear equations \( A \cdot X = B \) with a complex Hermitian matrix \( A \) using the factorization of \( A \):

if \( \text{uplo} = 'U' \), \[ A = U \cdot D \cdot U^H \]
if \( \text{uplo} = 'L' \), \[ A = L \cdot D \cdot L^H \]

where

- \( U \) and \( L \) are upper and lower triangular matrices with unit diagonal
- \( D \) is a Hermitian block-diagonal matrix.

The factorization is computed by \( \text{?hetrf} \).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (\( \text{LAPACK\_ROW\_MAJOR} \)) or column major (\( \text{LAPACK\_COL\_MAJOR} \)).

- **uplo**
  Must be \( 'U' \) or \( 'L' \).
  Indicates how the input matrix \( A \) has been factored:
  If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular factor \( U \) of the factorization \( A = U \cdot D \cdot U^H \).
  If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular factor \( L \) of the factorization \( A = L \cdot D \cdot L^H \).

- **n**
  The order of matrix \( A \); \( n \geq 0 \).

- **nrhs**
  The number of right-hand sides; \( nrhs \geq 0 \).

- **a**
  The array \( a \) of size \( \max(1, lda \times n) \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as computed by \( \text{?hetrf} \).

- **b**
  The array \( b \) of size \( \max(1, ldb \times nrhs) \) for column major layout and \( \max(1, ldb \times n) \) for row major layout contains the right-hand side matrix \( B \).

- **lda**
  The leading dimension of \( a \); \( lda \geq \max(1, n) \).

- **ldb**
  The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

- **ipiv**
  Array of size \( n \). The \( \text{ipiv} \) array contains details of the interchanges and the block structure of \( D \) as determined by \( \text{?hetrf} \).

Output Parameters

- **b**
  Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.
If $\text{info} = -i$, parameter $i$ had an illegal value.

**See Also**

?hetrf

**Matrix Storage Schemes**

**?sytrs_3**

Solves a system of linear equations $A \times X = B$ with a real or complex symmetric matrix.

```c
lapack_int LAPACKE_ssytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * A, lapack_int lda, const float * e, const lapack_int * ipiv, float * B, lapack_int ldb);

lapack_int LAPACKE_dsytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * A, lapack_int lda, const double * e, const lapack_int * ipiv, double * B, lapack_int ldb);

lapack_int LAPACKE_csytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);

lapack_int LAPACKE_zsytrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

**Description**

?sytrs_3 solves a system of linear equations $A \times X = B$ with a real or complex symmetric matrix $A$ using the factorization computed by ?sytrf_rk: $A = P \times U \times D \times (U^T) \times (P^T)$ or $A = P \times L \times D \times (L^T) \times (P^T)$, where $U$ (or $L$) is unit upper (or lower) triangular matrix, $U^T$ (or $L^T$) is the transpose of $U$ (or $L$), $P$ is a permutation matrix, $P^T$ is the transpose of $P$, and $D$ is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:
  - $'U'$: Upper triangular; the form is $A = P \times U \times D \times (U^T) \times (P^T)$.
  - $'L'$: Lower triangular; the form is $A = P \times L \times D \times (L^T) \times (P^T)$.

- **n**
  The order of the matrix $A$. $n \geq 0$.

- **nrhs**
  The number of right-hand sides; that is, the number of columns of the matrix $B$. $nrhs \geq 0$.

- **A**
  Array of size max$(1, \text{lda} \times n)$. Diagonal of the block diagonal matrix $D$ and factors $U$ or $L$ as computed by ?sytrf_rk:
  - Only diagonal elements of the symmetric block diagonal matrix $D$ on the diagonal of $A$; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of $D$ should be provided on entry in array $e$. —and—
• If $\text{uplo} = 'U'$, factor U in the superdiagonal part of A. If $\text{uplo} = 'L'$, factor L in the subdiagonal part of A.

$\text{lda}$

The leading dimension of the array $A$.

$\text{e}$

Array of size $n$. On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix $D$ with 1-by-1 or 2-by-2 diagonal blocks. If $\text{uplo} = 'U'$, $e(i) = D(i-1,i), i=2:N$, and $e(1)$ is not referenced. If $\text{uplo} = 'L'$, $e(i) = D(i+1,i), i=1:N-1$, and $e(n)$ is not referenced.

**NOTE** For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the $\text{uplo} = 'U'$ and $\text{uplo} = 'L'$ cases.

$\text{ipiv}$

Array of size $n$. Details of the interchanges and the block structure of $D$ as determined by ?sytrf_rk.

$B$

On entry, the right-hand side matrix $B$.

The size of $B$ is at least $\max(1, \text{ldb} \times \text{nrhs})$ for column-major layout and $\max(1, \text{ldb} \times n)$ for row-major layout.

$\text{ldb}$

The leading dimension of the array $B$. $\text{ldb} \geq \max(1, n)$ for column-major layout and $\text{ldb} \geq \text{nrhs}$ for row-major layout.

**Output Parameters**

$B$

On exit, the solution matrix $X$.

**Return Values**

This function returns a value $\text{info}$.

$= 0$: Successful exit.

$< 0$: If $\text{info} = -i$, the $i$th argument had an illegal value.

**?hetrs_3**

Solves a system of linear equations $A \times X = B$ with a complex Hermitian matrix using the factorization computed by ?hetrf_rk.

```c
lapack_int LAPACKE_chetrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);

lapack_int LAPACKE_zhetrs_3 (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

**Description**

?hetrs_3 solves a system of linear equations $A \times X = B$ with a complex Hermitian matrix $A$ using the factorization computed by ?hetrf_rk: $A = P^T U^* D(\text{U}^*)^* (P^T)$ or $A = P^T L^* D(\text{L}^*)^* (P^T)$, where $U$ (or $L$) is unit upper (or lower) triangular matrix, $U^*$ (or $L^*$) is the conjugate of $U$ (or $L$), $P$ is a permutation matrix, $P^T$ is the transpose of $P$, and $D$ is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

This algorithm uses Level 3 BLAS.
**Input Parameters**

`matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

`uplo` Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:
- `'U'`: Upper triangular; form is \( A = P \times U \times D \times (U^H) \times (P^T) \).
- `'L'`: Lower triangular; form is \( A = P \times L \times D \times (L^H) \times (P^T) \).

`n` The order of the matrix \( A \). \( n \geq 0 \).

`nrhs` The number of right-hand sides; that is, the number of columns in the matrix \( B \). \( nrhs \geq 0 \).

`A` Array of size \( \max(1, \text{lda} \times n) \). Diagonal of the block diagonal matrix \( D \) and factor \( U \) or \( L \) as computed by \(?hetrf_rk\):
- Only diagonal elements of the Hermitian block diagonal matrix \( D \) on the diagonal of \( A \); that is, \( D(k,k) = A(k,k) \). Superdiagonal (or subdiagonal) elements of \( D \) should be provided on entry in array \( e \).
- If `uplo` = `'U'`, factor \( U \) in the superdiagonal part of \( A \). If `uplo` = `'L'`, factor \( L \) in the subdiagonal part of \( A \).

`lda` The leading dimension of the array \( A \).

`e` Array of size \( n \). On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix \( D \) with 1-by-1 or 2-by-2 diagonal blocks. If `uplo` = `'U'`, \( e(i) = D(i-1,i), i=2:N \), and \( e(1) \) is not referenced. If `uplo` = `'L'`, \( e(i) = D(i+1,i), i=1:N-1 \), and \( e(n) \) is not referenced.

**NOTE** For 1-by-1 diagonal block \( D(k) \), where \( 1 \leq k \leq n \), the element \( e[k-1] \) is not referenced in both the `uplo` = `'U'` and `uplo` = `'L'` cases.

`ipiv` Array of size \( n \). Details of the interchanges and the block structure of \( D \) as determined by \(?hetrf_rk\).

`B` On entry, the right-hand side matrix \( B \).

The size of \( B \) is at least \( \max(1, \text{ldb} \times nrhs) \) for column-major layout and \( \max(1, \text{ldb} \times n) \) for row-major layout.

`ldb` The leading dimension of the array \( B \). \( ldb \geq \max(1, n) \) for column-major layout and \( ldb \geq nrhs \) for row-major layout.

**Output Parameters**

`B` On exit, the solution matrix \( X \).

**Return Values**

This function returns a value \( info \).

\( = 0 \): Successful exit.
< 0: If info = -i, the i-th argument had an illegal value.

`?sptrs`
Solves a system of linear equations with a UDU- or LDL-factored symmetric coefficient matrix using packed storage.

**Syntax**

```c
lapack_int LAPACKE_ssptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float * ap, const lapack_int * ipiv, float * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_dsptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double * ap, const lapack_int * ipiv, double * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_csptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * ap, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_zsptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * ap, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine solves for X the system of linear equations $A*X = B$ with a symmetric matrix $A$, given the Bunch-Kaufman factorization of $A$:

$$
\begin{align*}
\text{if } \text{uplo} &= \text{`U'}, & A &= U*Diag*D*U^T \\
\text{if } \text{uplo} &= \text{`L'}, & A &= L*Diag*D*L^T,
\end{align*}
$$

where $U$ and $L$ are upper and lower packed triangular matrices with unit diagonal and $D$ is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix $B$. You must supply the factor $U$ (or $L$) and the array `ipiv` returned by the factorization routine `?sptrf`.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be `U` or `L`.

  Indicates how the input matrix $A$ has been factored:

  If uplo = `U`, the array `ap` stores the packed factor $U$ of the factorization $A = U*Diag*D*U^T$. If uplo = `L`, the array `ap` stores the packed factor $L$ of the factorization $A = L*Diag*D*L^T$.

- **n**
  The order of matrix $A$; $n \geq 0$.

- **nrhs**
  The number of right-hand sides; $nrhs \geq 0$.

- **ipiv**
  Array, size at least max(1, n). The `ipiv` array, as returned by `?sptrf`. 


The dimension of array \( ap \) must be at least \( \max(1, n(n+1)/2) \). The array \( ap \) contains the factor \( U \) or \( L \), as specified by \textit{uplo}, in packed storage (see Matrix Storage Schemes).

The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the system of equations. The size of \( b \) is \( \max(1, \text{ldb} \times \text{nrhs}) \) for column major layout and \( \max(1, ldb \times n) \) for row major layout.

The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq \text{nrhs} \) for row major layout.

**Output Parameters**

\( b \) Overwritten by the solution matrix \( X \).

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

**Application Notes**

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \( (A + E)x = b \), where

\[
|E| \leq c(n)\varepsilon \|U\| \|D\| \|U^T\| F^2 \quad \text{or} \quad |E| \leq c(n)\varepsilon \|L\| \|D\| \|L^T\| F^2
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}_\infty (A, x) \varepsilon
\]

where \( \text{cond}_\infty (A, x) = \frac{\| A^{-1} \| \| A \| \| x \|_\infty}{\| x \|_\infty} \leq \frac{\| A^{-1} \| \| x \|_\infty}{\| A \|_\infty} = \kappa_\infty (A) \).

Note that \( \text{cond}_\infty (A, x) \) can be much smaller than \( \kappa_\infty (A) \).

The total number of floating-point operations for one right-hand side vector is approximately \( 2n^2 \) for real flavors or \( 8n^2 \) for complex flavors.

To estimate the condition number \( \kappa_\infty (A) \), call \textit{?spcon}.

To refine the solution and estimate the error, call \textit{?sprfs}.

**See Also**

Matrix Storage Schemes

\textit{?hptrs}

Solves a system of linear equations with a \( UDU^- \) or \( LDL^- \)-factored Hermitian coefficient matrix using packed storage.
Syntax

```c
lapack_int LAPACKE_chptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float * ap, const lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zhptrs (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double * ap, const lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine solves for $X$ the system of linear equations $AX = B$ with a Hermitian matrix $A$, given the Bunch-Kaufman factorization of $A$:

- If `uplo='U'`, $A = U*D*U^H$
- If `uplo='L'`, $A = L*D*L^H$

where $U$ and $L$ are upper and lower packed triangular matrices with unit diagonal and $D$ is a symmetric block-diagonal matrix. The system is solved with multiple right-hand sides stored in the columns of the matrix $B$.

You must supply to this routine the arrays `ap` (containing $U$ or $L$) and `ipiv` in the form returned by the factorization routine `?hptrf`.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo` Must be 'U' or 'L'.
- `n` The order of matrix $A$; $n \geq 0$.
- `nrhs` The number of right-hand sides; $nrhs \geq 0$.
- `ipiv` Array, size at least max(1, $n$). The `ipiv` array, as returned by `?hptrf`.
- `ap` The dimension of array `ap` must be at least max(1, $n(n+1)/2$). The array `ap` contains the factor $U$ or $L$, as specified by `uplo`, in packed storage (see Matrix Storage Schemes).
- `b` The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the system of equations. The size of $b$ is max(1, $ldb*nrhs$) for column major layout and max(1, $ldb*n$) for row major layout.
- `ldb` The leading dimension of $b$; $ldb \geq max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
Output Parameters

- \( b \)
  Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \((A + E)x = b\), where

\[
|E| \leq c(n)\varepsilon|U||D||L| \leq c(n)\varepsilon|P||U||D||L|
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n)\text{cond}(A, x)\varepsilon
\]

where \( \text{cond}(A, x) = \frac{||A^{-1}||A||x||_\infty}{||x||_\infty} \leq ||A^{-1}||_\infty ||A||_\infty = \kappa_\infty(A) \).

Note that \( \text{cond}(A, x) \) can be much smaller than \( \kappa_\infty(A) \).

The total number of floating-point operations for one right-hand side vector is approximately \( 8n^2 \) for complex flavors.

To estimate the condition number \( \kappa_\infty(A) \), call \( \text{hpcon} \).

To refine the solution and estimate the error, call \( \text{hprfs} \).

See Also

Matrix Storage Schemes

?trtrs

Solves a system of linear equations with a triangular coefficient matrix, with multiple right-hand sides.

Syntax

lapack_int LAPACKE_strtrs (int matrix_layout, char uplo, char trans, char diag, lapack_int n, lapack_int nrhs, const float * a, lapack_int lda, float * b, lapack_int ldb);

lapack_int LAPACKE_dtrtrs (int matrix_layout, char uplo, char trans, char diag, lapack_int n, lapack_int nrhs, const double * a, lapack_int lda, double * b, lapack_int ldb);

lapack_int LAPACKE_ctrtrs (int matrix_layout, char uplo, char trans, char diag, lapack_int n, lapack_int nrhs, const lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_ztrtrs (int matrix_layout, char uplo, char trans, char diag, lapack_int n, lapack_int nrhs, const lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);
Include Files
• mkl.h

Description
The routine solves for \( X \) the following systems of linear equations with a triangular matrix \( A \), with multiple right-hand sides stored in \( B \):

\[
A*X = B \quad \text{if } trans='N',
\]

\[
A^T*X = B \quad \text{if } trans='T',
\]

\[
A^H*X = B \quad \text{if } trans='C' \text{ (for complex matrices only)}.
\]

Input Parameters

\textit{matrix\_layout} \quad \text{Specifies whether matrix storage layout is row major (LAPACK\_ROW\_MAJOR) or column major (LAPACK\_COL\_MAJOR).}

\textit{uplo} \quad \text{Must be 'U' or 'L'.}

Indicates whether \( A \) is upper or lower triangular:

If \( \text{uplo} = 'U' \), then \( A \) is upper triangular.

If \( \text{uplo} = 'L' \), then \( A \) is lower triangular.

\textit{trans} \quad \text{Must be 'N' or 'T' or 'C'.}

If \( \text{trans} = 'N' \), then \( A*X = B \) is solved for \( X \).

If \( \text{trans} = 'T' \), then \( A^T*X = B \) is solved for \( X \).

If \( \text{trans} = 'C' \), then \( A^H*X = B \) is solved for \( X \).

\textit{diag} \quad \text{Must be 'N' or 'U'.}

If \( \text{diag} = 'N' \), then \( A \) is not a unit triangular matrix.

If \( \text{diag} = 'U' \), then \( A \) is unit triangular: diagonal elements of \( A \) are assumed to be 1 and not referenced in the array \( a \).

\textit{n} \quad \text{The order of \( A \); the number of rows in \( B \); } n \geq 0.

\textit{nrhs} \quad \text{The number of right-hand sides; } nrhs \geq 0.

\textit{a} \quad \text{The array } a \text{ contains the matrix } A.

The size of \( a \) is \text{max}(1, lda*n).

\textit{b} \quad \text{The array } b \text{ contains the matrix } B \text{ whose columns are the right-hand sides for the systems of equations.}

The size of \( b \) is \text{max}(1, ldb*nrhs) \text{ for column major layout and } \text{max}(1, ldb*n) \text{ for row major layout.}

\textit{lda} \quad \text{The leading dimension of } a; \quad \text{lda} \geq \text{max}(1, n).

\textit{ldb} \quad \text{The leading dimension of } b; \quad \text{ldb} \geq \text{max}(1, n) \text{ for column major layout and } \text{ldb} \geq nrhs \text{ for row major layout.}
Output Parameters

\( b \) 
Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \( (A + E) x = b \), where

\[
|E| \leq c(n) \varepsilon |A|
\]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision. If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond}(A, x) \varepsilon \text{ provided } c(n) \text{cond}(A, x) \varepsilon < 1
\]

where \( \text{cond}(A, x) = \| |A^{-1}|A| |x| |x|_\infty / |x|_\infty \leq \|A^{-1}|A|_\infty = \kappa_\infty(A) \).

Note that \( \text{cond}(A, x) \) can be much smaller than \( \kappa_\infty(A) \); the condition number of \( A^T \) and \( A^H \) might or might not be equal to \( \kappa_\infty(A) \).

The approximate number of floating-point operations for one right-hand side vector \( b \) is \( n^2 \) for real flavors and \( 4n^2 \) for complex flavors.

To estimate the condition number \( \kappa_\infty(A) \), call \( ?\text{trcon} \).

To estimate the error in the solution, call \( ?\text{trrfs} \).

See Also

Matrix Storage Schemes

\( ?\text{tptrs} \)

Solves a system of linear equations with a packed triangular coefficient matrix, with multiple right-hand sides.

Syntax

```c
lapack_int LAPACKE_stptrs (int matrix_layout , char uplo , char trans , char diag ,
                          lapack_int n , lapack_int nrhs , const float * ap , float * b , lapack_int ldb );
lapack_int LAPACKE_dtptrs (int matrix_layout , char uplo , char trans , char diag ,
                          lapack_int n , lapack_int nrhs , const double * ap , double * b , lapack_int ldb );
lapack_int LAPACKE_ctptrs (int matrix_layout , char uplo , char trans , char diag ,
                          lapack_int n , lapack_int nrhs , const lapack_complex_float * ap , lapack_complex_float *
                          b , lapack_int ldb );
lapack_int LAPACKE_ztptrs (int matrix_layout , char uplo , char trans , char diag ,
                          lapack_int n , lapack_int nrhs , const lapack_complex_double * ap ,
                          lapack_complex_double * b , lapack_int ldb );
```

495
Include Files
- mkl.h

Description
The routine solves for $X$ the following systems of linear equations with a packed triangular matrix $A$, with multiple right-hand sides stored in $B$:

- $A*X = B$ if $trans='N'$,
- $A^T*X = B$ if $trans='T'$,
- $A^H*X = B$ if $trans='C'$ (for complex matrices only).

Input Parameters

- $matrix\_layout$: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- $uplo$: Must be 'U' or 'L'. Indicates whether $A$ is upper or lower triangular:
  - If $uplo = 'U'$, then $A$ is upper triangular.
  - If $uplo = 'L'$, then $A$ is lower triangular.
- $trans$: Must be 'N' or 'T' or 'C'.
  - If $trans = 'N'$, then $A*X = B$ is solved for $X$.
  - If $trans = 'T'$, then $A^T*X = B$ is solved for $X$.
  - If $trans = 'C'$, then $A^H*X = B$ is solved for $X$.
- $diag$: Must be 'N' or 'U'.
  - If $diag = 'N'$, then $A$ is not a unit triangular matrix.
  - If $diag = 'U'$, then $A$ is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array $ap$.
- $n$: The order of $A$; the number of rows in $B$; $n \geq 0$.
- $nrhs$: The number of right-hand sides; $nrhs \geq 0$.
- $ap$: The dimension of array $ap$ must be at least max(1,$n(n+1)/2$). The array $ap$ contains the matrix $A$ in packed storage (see Matrix Storage Schemes).
- $b$: The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the system of equations.
  - The size of $b$ is max(1, $ldb*nrhs$) for column major layout and max(1, $ldb*n$) for row major layout.
- $ldb$: The leading dimension of $b$; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
Output Parameters

\[ b \]

Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

For each right-hand side \( b \), the computed solution is the exact solution of a perturbed system of equations \((A + E)x = b\), where

\[ |E| \leq c(n)\varepsilon |A| \]

\( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[ \frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond} (A, x)\varepsilon \text{ provided } c(n) \text{cond} (A, x)\varepsilon < 1 \]

where \( \text{cond} (A, x) = \| |A^{-1}| |A| |x| \| / \| |x|\| \leq | |A^{-1}| |A| \| \|n = \kappa_n (A) \).

Note that \( \text{cond} (A, x) \) can be much smaller than \( \kappa_n (A) \); the condition number of \( A^T \) and \( A^H \) might or might not be equal to \( \kappa_n (A) \).

The approximate number of floating-point operations for one right-hand side vector \( b \) is \( n^2 \) for real flavors and \( 4n^2 \) for complex flavors.

To estimate the condition number \( \kappa_n (A) \), call \(?tpcon\).

To estimate the error in the solution, call \(?tprfs\).

See Also

Matrix Storage Schemes

\(?tbtrs\)

Solves a system of linear equations with a band triangular coefficient matrix, with multiple right-hand sides.

Syntax

\[
\begin{align*}
lapack_int \text{LAPACKE_stbtrs} ( \text{int matrix_layout} , \text{char uplo} , \text{char trans} , \text{char diag} , 
\text{lapack_int n} , \text{lapack_int kd} , \text{lapack_int nrhs} , \text{const float * ab} , \text{lapack_int ldab} , 
\text{float * b} , \text{lapack_int ldb}) ; \\
lapack_int \text{LAPACKE_dtbtrs} ( \text{int matrix_layout} , \text{char uplo} , \text{char trans} , \text{char diag} , 
\text{lapack_int n} , \text{lapack_int kd} , \text{lapack_int nrhs} , \text{const double * ab} , \text{lapack_int ldab} , 
\text{double * b} , \text{lapack_int ldb}) ; \\
lapack_int \text{LAPACKE_ctbtrs} ( \text{int matrix_layout} , \text{char uplo} , \text{char trans} , \text{char diag} , 
\text{lapack_int n} , \text{lapack_int kd} , \text{lapack_int nrhs} , \text{const lapack_complex_float * ab} , 
\text{lapack_int ldab} , \text{lapack_complex_float * b} , \text{lapack_int ldb}) ;
\end{align*}
\]
lapack_int LAPACKE_ztbtrs (int matrix_layout, char uplo, char trans, char diag, 
lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_double * ab, 
lapack_int ldab, lapack_complex_double * b, lapack_int ldb);

Include Files
• mkl.h

Description
The routine solves for $X$ the following systems of linear equations with a band triangular matrix $A$, with 
multiple right-hand sides stored in $B$:

\[
\begin{align*}
A \cdot X &= B & \text{if } \text{trans='N'}, \\
A^T \cdot X &= B & \text{if } \text{trans='T'}, \\
A^H \cdot X &= B & \text{if } \text{trans='C'} \text{ (for complex matrices only)}. 
\end{align*}
\]

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major 
(LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo Must be 'U' or 'L'. 
Indicates whether $A$ is upper or lower triangular: 
If uplo = 'U', then $A$ is upper triangular. 
If uplo = 'L', then $A$ is lower triangular.

trans Must be 'N' or 'T' or 'C'. 
If trans = 'N', then $A \cdot X = B$ is solved for $X$. 
If trans = 'T', then $A^T \cdot X = B$ is solved for $X$. 
If trans = 'C', then $A^H \cdot X = B$ is solved for $X$.

diag Must be 'N' or 'U'. 
If diag = 'N', then $A$ is not a unit triangular matrix. 
If diag = 'U', then $A$ is unit triangular: diagonal elements are 
assumed to be 1 and not referenced in the array ab.

n The order of $A$; the number of rows in $B$; $n \geq 0$.

kd The number of superdiagonals or subdiagonals in the matrix $A$; $kd \geq 0$.

nrhs The number of right-hand sides; $nrhs \geq 0$.

ab The array $ab$ contains the matrix $A$ in band storage form.
The size of $ab$ must be $\max(1, \text{ldab} \cdot n)$

b The array $b$ contains the matrix $B$ whose columns are the right-hand 
sides for the systems of equations. 
The size of $b$ is $\max(1, \text{ldb} \cdot nrhs)$ for column major layout and 
$\max(1, \text{ldb} \cdot n)$ for row major layout.
ldab  The leading dimension of ab; \( ldab \geq kd + 1 \).

ldb  The leading dimension of b; \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Output Parameters

b  Overwritten by the solution matrix X.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, parameter i had an illegal value.

Application Notes

For each right-hand side b, the computed solution is the exact solution of a perturbed system of equations \((A + E)x = b\), where

\[ \|E\| \leq c(n)\epsilon\|A\| \]

\( c(n) \) is a modest linear function of \( n \), and \( \epsilon \) is the machine precision. If \( x_0 \) is the true solution, the computed solution \( x \) satisfies this error bound:

\[
\frac{\|x - x_0\|_\infty}{\|x\|_\infty} \leq c(n) \text{cond} (A, x) \epsilon \quad \text{provided} \quad c(n) \text{cond} (A, x) \epsilon < 1
\]

where \( \text{cond} (A, x) = \frac{||A^{-1}||A||x||_\infty}{||x||_\infty} \leq ||A^{-1}||_\infty \frac{||A||_\infty}{||A||_\infty} = \kappa_\infty (A) \).

Note that \( \text{cond} (A, x) \) can be much smaller than \( \kappa_\infty (A) \); the condition number of \( A^T \) and \( A^H \) might or might not be equal to \( \kappa_\infty (A) \).

The approximate number of floating-point operations for one right-hand side vector \( b \) is \( 2n^*kd \) for real flavors and \( 8n^*kd \) for complex flavors.

To estimate the condition number \( \kappa_\infty (A) \), call \texttt{?tbcon}.

To estimate the error in the solution, call \texttt{?tbrfs}.

See Also

Matrix Storage Schemes

Estimating the Condition Number: LAPACK Computational Routines

This section describes the LAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations (see Error Analysis). Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the reciprocal condition number.

?gecon

Estimates the reciprocal of the condition number of a general matrix in the 1-norm or the infinity-norm.
**Syntax**

```c
lapack_int LAPACKE_sgecon( int matrix_layout, char norm, lapack_int n, const float* a, lapack_int lda, float anorm, float* rcond );

lapack_int LAPACKE_dgecon( int matrix_layout, char norm, lapack_int n, const double* a, lapack_int lda, double anorm, double* rcond );

lapack_int LAPACKE_cgecon( int matrix_layout, char norm, lapack_int n, const lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );

lapack_int LAPACKE_zgecon( int matrix_layout, char norm, lapack_int n, const lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );
```

**Include Files**
- mkl.h

**Description**

The routine estimates the reciprocal of the condition number of a general matrix $A$ in the 1-norm or infinity-norm:

- $\kappa_1(A) = \|A\|_1\|A^{-1}\|_1 = \kappa_1(A^T) = \kappa_1(A^H)$
- $\kappa_\infty(A) = \|A\|_\infty\|A^{-1}\|_\infty = \kappa_\infty(A^T) = \kappa_1(A^H)$.

An estimate is obtained for $\|A^{-1}\|$, and the reciprocal of the condition number is computed as $rcond = 1 / (\|A\|_1 \|A^{-1}\|_1)$.

Before calling this routine:
- compute $anorm$ (either $\|A\|_1 = \max_i \sum_j |a_{ij}|$ or $\|A\|_\infty = \max_j \sum_i |a_{ij}|$)
- call `?getrf` to compute the $LU$ factorization of $A$.

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).
- `norm` Must be '1' or 'O' or 'I'.
  - If `norm` = '1' or 'O', then the routine estimates the condition number of matrix $A$ in 1-norm.
  - If `norm` = 'I', then the routine estimates the condition number of matrix $A$ in infinity-norm.
- `n` The order of the matrix $A$; $n \geq 0$.
- `a` The array $a$ contains the $LU$-factored matrix $A$, as returned by `?getrf`.
- `anorm` The norm of the original matrix $A$ (see `Description`).
- `lda` The leading dimension of $a$; $lda \geq \max(1, n)$. 
Output Parameters

\( rcond \)

An estimate of the reciprocal of the condition number. The routine sets \( rcond = 0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( rcond \) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The computed \( rcond \) is never less than \( r \) (the reciprocal of the true condition number) and in practice is nearly always less than 10\( r \). A call to this routine involves solving a number of systems of linear equations \( A^* x = b \) or \( A^H x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors and \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

?gbcon

Estimates the reciprocal of the condition number of a band matrix in the 1-norm or the infinity-norm.

Syntax

lapack_int LAPACKE_sgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl, lapack_int ku, const float* ab, lapack_int ldab, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_dgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl, lapack_int ku, const double* ab, lapack_int ldab, const lapack_int* ipiv, double anorm, double* rcond );

lapack_int LAPACKE_cgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, const lapack_int* ipiv, float anorm, float* rcond );

lapack_int LAPACKE_zgbcon( int matrix_layout, char norm, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, const lapack_int* ipiv, double anorm, double* rcond );

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a general band matrix \( A \) in the 1-norm or infinity-norm:

\[ \kappa_1(A) = ||A||_1 ||A^{-1}||_1 = \kappa_1(A^T) = \kappa_1(A^H) \]

\[ \kappa_\infty(A) = ||A||_\infty ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H). \]
An estimate is obtained for \(||A^{-1}|||\), and the reciprocal of the condition number is computed as \(rcond = 1 / (||A|| ||A^{-1}||)\).

Before calling this routine:
- compute \(anorm\) (either \(||A||_1 = \max \sum |a_{ij}|\) or \(||A||_\infty = \max \sum |a_{ij}|\))
- call \(?gbtrf\) to compute the \(LU\) factorization of \(A\).

**Input Parameters**

- \(matrix\_layout\)  
  Specifies whether matrix storage layout is row major (\(\text{LAPACK\_ROW\_MAJOR}\)) or column major (\(\text{LAPACK\_COL\_MAJOR}\)).

- \(norm\)
  Must be ‘1’ or ‘0’ or ‘I’.
  - If \(norm = '1'\) or ‘0’, then the routine estimates the condition number of matrix \(A\) in 1-norm.
  - If \(norm = 'I'\), then the routine estimates the condition number of matrix \(A\) in infinity-norm.

- \(n\)
  The order of the matrix \(A\); \(n \geq 0\).

- \(kl\)
  The number of subdiagonals within the band of \(A\); \(kl \geq 0\).

- \(ku\)
  The number of superdiagonals within the band of \(A\); \(ku \geq 0\).

- \(ldab\)
  The leading dimension of the array \(ab\). \((ldab \geq 2*kl + ku + 1)\).

- \(ipiv\)
  Array, size at least \(\max(1, n)\). The \(ipiv\) array, as returned by \(?gbtrf\).

- \(ab\)
  The array \(ab\) of size \(\max(1, ldab*n)\) contains the factored band matrix \(A\), as returned by \(?gbtrf\).

- \(anorm\)
  The norm of the original matrix \(A\)(see Description).

**Output Parameters**

- \(rcond\)
  An estimate of the reciprocal of the condition number. The routine sets \(rcond = 0\) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \(rcond\) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

**Return Values**

This function returns a value \(info\).

- If \(info = 0\), the execution is successful.
- If \(info = -i\), parameter \(i\) had an illegal value.

**Application Notes**

The computed \(rcond\) is never less than \(r\) (the reciprocal of the true condition number) and in practice is nearly always less than \(10r\). A call to this routine involves solving a number of systems of linear equations \(A^*x = b\) or \(A^Hx = b\); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \(2n(ku + 2kl)\) floating-point operations for real flavors and \(8n(ku + 2kl)\) for complex flavors.
See Also
Matrix Storage Schemes

?gtcon
Estimates the reciprocal of the condition number of a tridiagonal matrix.

Syntax

```c
lapack_int LAPACKE_sgtcon( char norm, lapack_int n, const float* dl, const float* d, const float* du, const float* du2, const lapack_int* ipiv, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_dgtcon( char norm, lapack_int n, const double* dl, const double* d, const double* du, const double* du2, const lapack_int* ipiv, double anorm, double* rcond );
```

```c
lapack_int LAPACKE_cgtcon( char norm, lapack_int n, const lapack_complex_float* dl, const lapack_complex_float* d, const lapack_complex_float* du, const lapack_complex_float* du2, const lapack_int* ipiv, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_zgtcon( char norm, lapack_int n, const lapack_complex_double* dl, const lapack_complex_double* d, const lapack_complex_double* du, const lapack_complex_double* du2, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files
- mkl.h

Description

The routine estimates the reciprocal of the condition number of a real or complex tridiagonal matrix A in the 1-norm or infinity-norm:

- \( \kappa_1(A) = \frac{||A|| \cdot ||A^{-1}||}{1} \)
- \( \kappa_\infty(A) = \frac{||A|| \cdot ||A^{-1}||}{\infty} \)

An estimate is obtained for \( ||A^{-1}|| \), and the reciprocal of the condition number is computed as \( rcond = \frac{1}{||A|| \cdot ||A^{-1}||} \).

Before calling this routine:
- compute \( \text{anorm} \) (either \( ||A||_1 = \max_j \sum_i |a_{ij}| \) or \( ||A||_\infty = \max_i \sum_j |a_{ij}| \))
- call ?gttrf to compute the LU factorization of A.

Input Parameters

- **norm**
  - Must be '1' or 'O' or 'I'.
  - If \( \text{norm} = '1' \) or '0', then the routine estimates the condition number of matrix A in 1-norm.
  - If \( \text{norm} = 'I' \), then the routine estimates the condition number of matrix A in infinity-norm.

- **n**
  - The order of the matrix A; \( n \geq 0 \).

- **dl, d, du, du2**
  - Arrays: \( dl(n-1), d(n), du(n-1), du2(n-2) \).
  - The array \( dl \) contains the \((n - 1)\) multipliers that define the matrix L from the LU factorization of A as computed by ?gttrf.
The array \( d \) contains the \( n \) diagonal elements of the upper triangular matrix \( U \) from the LU factorization of \( A \).

The array \( du \) contains the \((n - 1)\) elements of the first superdiagonal of \( U \).

The array \( du2 \) contains the \((n - 2)\) elements of the second superdiagonal of \( U \).

\( ipiv \)

Array, size \( (n) \). The array of pivot indices, as returned by \?gttrf.

\( anorm \)

The norm of the original matrix \( A \)(see Description).

Output Parameters

\( rcond \)

An estimate of the reciprocal of the condition number. The routine sets \( rcond=0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( rcond \) is small compared to \( 1.0 \), for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The computed \( rcond \) is never less than \( r \) (the reciprocal of the true condition number) and in practice is nearly always less than \( 10r \). A call to this routine involves solving a number of systems of linear equations \( A^*x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors and \( 8n^2 \) for complex flavors.

?pocon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix.

Syntax

lapack_int LAPACKE_spocon( int matrix_layout, char uplo, lapack_int n, const float* a, lapack_int lda, float anorm, float* rcond );
lapack_int LAPACKE_dpocon( int matrix_layout, char uplo, lapack_int n, const double* a, lapack_int lda, double anorm, double* rcond );
lapack_int LAPACKE_cpocon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* a, lapack_int lda, float anorm, float* rcond );
lapack_int LAPACKE_zpocon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* a, lapack_int lda, double anorm, double* rcond );

Include Files

- mkl.h
### Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite matrix $A$:

$$\kappa_1(A) = ||A||_1 ||A^{-1}||_1$$  
(since $A$ is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| ||A^{-1}||)$.

Before calling this routine:
- compute $anorm$ (either $||A||_1 = \max_j \Sigma_i |a_{ij}|$ or $||A||_\infty = \max_i \Sigma_j |a_{ij}|$)
- call $?potrf$ to compute the Cholesky factorization of $A$.

### Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'. Indicates how the input matrix $A$ has been factored:
  - If $\text{uplo} = 'U'$, $A$ is factored as $A = U^T U$ for real flavors or $A = UH^* U$ for complex flavors, and $U$ is stored.
  - If $\text{uplo} = 'L'$, $A$ is factored as $A = LL^T$ for real flavors or $A = L^* LH$ for complex flavors, and $L$ is stored.
- **n**: The order of the matrix $A$; $n \geq 0$.
- **a**: The array $a$ of size $\max(1, lda*n)$ contains the factored matrix $A$, as returned by $?potrf$.
- **lda**: The leading dimension of $a$; $lda \geq \max(1, n)$.
- **anorm**: The norm of the original matrix $A$ (see Description).

### Output Parameters

- **rcond**: An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

### Return Values

This function returns a value $\text{info}$.

- If $\text{info} = 0$, the execution is successful.
- If $\text{info} = -i$, parameter $i$ had an illegal value.

### Application Notes

The computed $rcond$ is never less than $r$ (the reciprocal of the true condition number) and in practice is nearly always less than 10$r$. A call to this routine involves solving a number of systems of linear equations $A^* x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors and $8n^2$ for complex flavors.
See Also
Matrix Storage Schemes

?ppcon
Estimates the reciprocal of the condition number of a
packed symmetric (Hermitian) positive-definite
matrix.

Syntax

```c
lapack_int LAPACKE_sppcon( int matrix_layout, char uplo, lapack_int n, const float* ap, 
float anorm, float* rcond );
```

```c
lapack_int LAPACKE_dppcon( int matrix_layout, char uplo, lapack_int n, const double* 
ap, double anorm, double* rcond );
```

```c
lapack_int LAPACKE_cppcon( int matrix_layout, char uplo, lapack_int n, const 
lapack_complex_float* ap, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_zppcon( int matrix_layout, char uplo, lapack_int n, const 
lapack_complex_double* ap, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a packed symmetric (Hermitian) positive-
definite matrix $A$:

$$ \kappa_1(A) = \frac{||A||_1}{||A^{-1}||_1} $$

(since $A$ is symmetric or Hermitian, $\kappa_\infty(A) = \kappa_1(A)$).

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 
1 / (||A|| \cdot ||A^{-1}||)$.

Before calling this routine:

- compute $anorm$ (either $||A||_1 = \max_i \sum_j |a_{ij}|$ or $||A||_\infty = \max_j \sum_i |a_{ij}|$)
- call ?pptrf to compute the Cholesky factorization of $A$.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates how the input matrix $A$ has been factored:
  If uplo = 'U', $A$ is factored as $A = U^*U$ for real flavors or $A = U^H*U$ for complex flavors, and $U$ is stored.
  If uplo = 'L', $A$ is factored as $A = L*L^*$ for real flavors or $A = L^*L^H$ for complex flavors, and $L$ is stored.

- **n**
  The order of the matrix $A$; $n \geq 0$.

- **ap**
  The array $ap$ contains the packed factored matrix $A$, as returned by ?pptrf. The dimension of $ap$ must be at least $\max(1,n(n+1)/2)$.
  The norm of the original matrix $A$ (see Description).
Output Parameters

\( rcond \)

An estimate of the reciprocal of the condition number. The routine sets \( rcond = 0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( rcond \) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The computed \( rcond \) is never less than \( r \) (the reciprocal of the true condition number) and in practice is nearly always less than 10\( r \). A call to this routine involves solving a number of systems of linear equations \( A^T x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors and \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

\(?pbcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix.

Syntax

\begin{verbatim}
lapack_int LAPACKE_spbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const float* ab, lapack_int ldab, float anorm, float* rcond );
lapack_int LAPACKE_dpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const double* ab, lapack_int ldab, double anorm, double* rcond );
lapack_int LAPACKE_cpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const lapack_complex_float* ab, lapack_int ldab, float anorm, float* rcond );
lapack_int LAPACKE_zpbcon( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const lapack_complex_double* ab, lapack_int ldab, double anorm, double* rcond );
\end{verbatim}

Include Files

• mkl.h

Description

The routine estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite band matrix \( A \):

\[ \kappa_1(A) = \| A \|_1 / \| A^{-1} \|_1 \]  
(since \( A \) is symmetric or Hermitian, \( \kappa_\infty(A) = \kappa_1(A) \)).

An estimate is obtained for \( \| A^{-1} \|, \) and the reciprocal of the condition number is computed as \( rcond = 1 / (\| A \| / \| A^{-1} \|) \).

Before calling this routine:

• compute \( anorm \) (either \( \| A \|_1 = \max_j \sum_i |a_{ij}| \) or \( \| A \|_\infty = \max_i \sum_j |a_{ij}| \))
• call ?pbtrf to compute the Cholesky factorization of A.

**Input Parameters**

*matrix_layout* Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

*uplo* Must be 'U' or 'L'.

Indicates how the input matrix A has been factored:

If *uplo* = 'U', A is factored as $A = U^T \cdot U$ for real flavors or $A = U^H \cdot U$ for complex flavors, and $U$ is stored.

If *uplo* = 'L', A is factored as $A = L \cdot L^T$ for real flavors or $A = L \cdot L^H$ for complex flavors, and $L$ is stored.

*n* The order of the matrix $A$; $n \geq 0$.

*kd* The number of superdiagonals or subdiagonals in the matrix $A$; $kd \geq 0$.

*ldab* The leading dimension of the array $ab$. ($ldab \geq kd + 1$).

*ab* The array $ab$ of size max(1, $ldab \cdot n$) contains the factored matrix $A$ in band form, as returned by ?pbtrf.

*anorm* The norm of the original matrix $A$ (see Description).

**Output Parameters**

*rcond* An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

**Return Values**

This function returns a value *info*.

If *info*=0, the execution is successful.

If *info* = -i, parameter i had an illegal value.

**Application Notes**

The computed $rcond$ is never less than $r$ (the reciprocal of the true condition number) and in practice is nearly always less than 10$r$. A call to this routine involves solving a number of systems of linear equations $A^* \cdot x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4 \cdot n (kd + 1)$ floating-point operations for real flavors and $16 \cdot n (kd + 1)$ for complex flavors.

**See Also**

Matrix Storage Schemes

?ptcon

Estimates the reciprocal of the condition number of a symmetric (Hermitian) positive-definite tridiagonal matrix.
Syntax

```c
lapack_int LAPACKE_sptcon( lapack_int n, const float* d, const float* e, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_dptcon( lapack_int n, const double* d, const double* e, double anorm, double* rcond );
```

```c
lapack_int LAPACKE_cptcon( lapack_int n, const float* d, const lapack_complex_float* e, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_zptcon( lapack_int n, const double* d, const lapack_complex_double* e, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine computes the reciprocal of the condition number (in the 1-norm) of a real symmetric or complex Hermitian positive-definite tridiagonal matrix using the factorization $A = L \cdot D \cdot L^T$ for real flavors and $A = L \cdot D \cdot L^H$ for complex flavors or $A = U^T \cdot D \cdot U$ for real flavors and $A = U^H \cdot D \cdot U$ for complex flavors computed by `$pttrf$`: 

$$\kappa_1(A) = ||A||_1 \cdot ||A^{-1}||_1$$

(since $A$ is symmetric or Hermitian, $\kappa_1(A) = \kappa_1(A)$).

The norm $||A^{-1}||$ is computed by a direct method, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \cdot ||A^{-1}||)$.

Before calling this routine:

- compute $anorm$ as $||A||_1 = \max_i \sum_j |a_{ij}|$
- call `$pttrf$` to compute the factorization of $A$.

Input Parameters

- $n$ The order of the matrix $A$; $n \geq 0$.
- $d$ Arrays, dimension $(n)$. The array $d$ contains the $n$ diagonal elements of the diagonal matrix $D$ from the factorization of $A$, as computed by `$pttrf$`.
- $e$ Array, size $(n - 1)$. Contains off-diagonal elements of the unit bidiagonal factor $U$ or $L$ from the factorization computed by `$pttrf$`.
- $anorm$ The 1-norm of the original matrix $A$ (see Description).

Output Parameters

- $rcond$ An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to $1.0$, for the working precision, the matrix may be poorly conditioned or even singular.
Return Values
This function returns a value info.

If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.

Application Notes
The computed rcond is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than 10r. A call to this routine involves solving a number of systems of linear equations \( A^T x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately 4*n*(kd + 1) floating-point operations for real flavors and 16*n*(kd + 1) for complex flavors.

?sycon
Estimates the reciprocal of the condition number of a symmetric matrix.

Syntax

```c
lapack_int LAPACKE_ssycon( int matrix_layout, char uplo, lapack_int n, const float* a, lapack_int lda, const lapack_int* ipiv, float anorm, float* rcond );
```
```c
lapack_int LAPACKE_dsycon( int matrix_layout, char uplo, lapack_int n, const double* a, lapack_int lda, const lapack_int* ipiv, double anorm, double* rcond );
```
```c
lapack_int LAPACKE_csycon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm, float* rcond );
```
```c
lapack_int LAPACKE_zsycon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files
- mkl.h

Description
The routine estimates the reciprocal of the condition number of a symmetric matrix \( A \):

\[
\kappa_1(A) = ||A||_1 ||A^{-1}||_1 \text{ (since } A \text{ is symmetric, } \kappa_\infty(A) = \kappa_1(A)\).
\]

An estimate is obtained for ||\( A^{-1} ||_1 \), and the reciprocal of the condition number is computed as \( rcond = 1 / (||A||_1 ||A^{-1}||_1) \).

Before calling this routine:
- compute anorm (either ||A||_1 = max_i |a_i| or ||A||_\infty = max_j |a_j|)
- call ?sytrf to compute the factorization of A.

Input Parameters

- matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- uplo Must be 'U' or 'L'.

Indicates how the input matrix \( A \) has been factored:

If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular factor \( U \) of the factorization \( A = U*D*U^T \).

If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular factor \( L \) of the factorization \( A = L*D*L^T \).

\( n \)

The order of matrix \( A \); \( n \geq 0 \).

\( a \)

The array \( a \) of size \( \max(1, \text{lda} \times n) \) contains the factored matrix \( A \), as returned by \(?sytrf\).

\( \text{lda} \)

The leading dimension of \( a \); \( \text{lda} \geq \max(1, n) \).

\( \text{ipiv} \)

Array, size at least \( \max(1, n) \).

The array \( \text{ipiv} \), as returned by \(?sytrf\).

\( \text{anorm} \)

The norm of the original matrix \( A \) (see Description).

**Output Parameters**

\( \text{rcond} \)

An estimate of the reciprocal of the condition number. The routine sets \( \text{rcond} = 0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( \text{rcond} \) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

**Application Notes**

The computed \( \text{rcond} \) is never less than \( r \) (the reciprocal of the true condition number) and in practice is nearly always less than 10\( r \). A call to this routine involves solving a number of systems of linear equations \( A^*x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors and \( 8n^2 \) for complex flavors.

**See Also**

**Matrix Storage Schemes**

\(?sycon_3\)

Estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix \( A \) using the factorization computed by \(?sytrf_rk\).

```c
lapack_int LAPACK_ssycon_3 (int matrix_layout, char uplo, lapack_int n, const float * A, lapack_int lda, const float * e, const lapack_int * ipiv, float anorm, float * rcond);

lapack_int LAPACK_dsycon_3 (int matrix_layout, char uplo, lapack_int n, const double * A, lapack_int lda, const double * e, const lapack_int * ipiv, double anorm, double * rcond);
```
lapack_int LAPACKE_csycon_3 (int matrix_layout, char uplo, lapack_int n, const lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const lapack_int * ipiv, float anorm, float * rcond);

lapack_int LAPACKE_zsycon_3 (int matrix_layout, char uplo, lapack_int n, const lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const lapack_int * ipiv, double anorm, double * rcond);

Description

`?sycon_3` estimates the reciprocal of the condition number (in the 1-norm) of a real or complex symmetric matrix A using the factorization computed by `?sytrf_rk`. A = P*U*D*(Uᵀ)*(Pᵀ) or A = P*L*D*(Lᵀ)*(Pᵀ), where U (or L) is unit upper (or lower) triangular matrix, Uᵀ (or Lᵀ) is the transpose of U (or L), P is a permutation matrix, Pᵀ is the transpose of P, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

An estimate is obtained for norm(inv(A)), and the reciprocal of the condition number is computed as rcond = 1 / (anorm * norm(inv(A))).

This routine uses BLAS3 solver `?sytrs_3`.

Input Parameters

* **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

* **uplo**
  Specifies whether the details of the factorization are stored as an upper or lower triangular matrix:
  - 'U': Upper triangular. The form is A = P*U*D*(Uᵀ)*(Pᵀ).
  - 'L': Lower triangular. The form is A = P*L*D*(Lᵀ)*(Pᵀ).

* **n**
  The order of the matrix A. n ≥ 0.

* **A**
  Array of size max(1, lda*n). Diagonal of the block diagonal matrix D and factors U or L as computed by `?sytrf_rk`:
  - Only diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, D(k,k) = A(k,k). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array e).
  - If uplo = 'U', factor U in the superdiagonal part of A. If uplo = 'L', factor L in the subdiagonal part of A.

* **lda**
  The leading dimension of the array A.

* **e**
  Array of size n. On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If uplo = 'U', e(i) = D(i-1,i), i = 2:N, and e(1) is not referenced. If uplo = 'L', e(i) = D(i+1,i), i = 1:N-1, and e(n) is not referenced.

**NOTE** For 1-by-1 diagonal block D(k), where 1 ≤ k ≤ n, the element e[k-1] is not referenced in both the uplo = 'U' and uplo = 'L' cases.
**ipiv**

Array of size \( n \). Details of the interchanges and the block structure of \( D \) as determined by \(?sytrf_rk\).

**anorm**

The 1-norm of the original matrix \( A \).

**Output Parameters**

**rcond**

The reciprocal of the condition number of the matrix \( A \), computed as \( rcond = 1/(anorm \times AINVNM) \), where \( AINVNM \) is an estimate of the 1-norm of \( \text{inv}(A) \) computed in this routine.

**Return Values**

This function returns a value \( info \).

- \( = 0 \): Successful exit.
- \(< 0\): If \( info = -i \), the \( i \)th argument had an illegal value.

**?hecon**

Estimates the reciprocal of the condition number of a Hermitian matrix.

**Syntax**

```c
lapack_int LAPACKE_checon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* a, lapack_int lda, const lapack_int* ipiv, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_zhecon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* a, lapack_int lda, const lapack_int* ipiv, double anorm, double* rcond );
```

**Include Files**

- mkl.h

**Description**

The routine estimates the reciprocal of the condition number of a Hermitian matrix \( A \):

\[
\kappa_1(A) = \| \|A|\|_1 / \| \|A^{-1}|\|_1 \quad (\text{since } A \text{ is Hermitian, } \kappa_\infty(A) = \kappa_1(A))
\]

Before calling this routine:

- compute \( anorm \) (either \( \| |A| \|_1 = \max_j \sum_i |a_{ij}| \) or \( \| |A| \|_\infty = \max_i \sum_j |a_{ij}| \))
- call \(?hetrf\) to compute the factorization of \( A \).

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'.

Indicates how the input matrix \( A \) has been factored:

- If \( uplo = 'U' \), the array \( a \) stores the upper triangular factor \( U \) of the factorization \( A = U*D*U^H \).
If `uplo = 'L'`, the array `a` stores the lower triangular factor `L` of the factorization `A = L*D*L^H`.

`n`

The order of matrix `A`; `n ≥ 0`.

`a`

The array `a` of size `max(1, lda*n)` contains the factored matrix `A`, as returned by `?hetrf`.

`lda`

The leading dimension of `a`; `lda ≥ max(1, n)`.

`ipiv`

Array, size at least `max(1, n)`.

The array `ipiv`, as returned by `?hetrf`.

`anorm`

The norm of the original matrix `A` (see Description).

### Output Parameters

`rcond`

An estimate of the reciprocal of the condition number. The routine sets `rcond = 0` if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime `rcond` is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

### Return Values

This function returns a value `info`.

- If `info = 0`, the execution is successful.
- If `info = -i`, parameter `i` had an illegal value.

### Application Notes

The computed `rcond` is never less than `r` (the reciprocal of the true condition number) and in practice is nearly always less than `10r`. A call to this routine involves solving a number of systems of linear equations `A*x = b`; the number is usually 5 and never more than 11. Each solution requires approximately `8n^2` floating-point operations.

### See Also

Matrix Storage Schemes

`?hecon_3`

Estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix `A`.

```c
lapack_int LAPACK_checon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv, float anorm, float * rcond);
```

```c
lapack_int LAPACK_zhecon_3 (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv, double anorm, double * rcond);
```

### Description

`?hecon_3` estimates the reciprocal of the condition number (in the 1-norm) of a complex Hermitian matrix `A` using the factorization computed by `?hetrf_rk`: `A = P*U*D*(U^H)*(P^T)` or `A = P*L*D*(L^H)*(P^T)`, where `U` (or `L`) is unit upper (or lower) triangular matrix, `U^H` (or `L^H`) is the conjugate of `U` (or `L`), `P` is a permutation...
matrix, $P^T$ is the transpose of $P$, and $D$ is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks. An estimate is obtained for $\text{norm}(\text{inv}(A))$, and the reciprocal of the condition number is computed as $rcond = 1 / (\text{anorm} \times \text{norm}(\text{inv}(A)))$.

This routine uses BLAS3 solver `?hetrs_3`.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).
- **uplo**: Specifies whether the details of the factorization are stored as an upper or lower triangular matrix: `$'U'$`: Upper triangular, form is $A = P*U*D*(U^H)*(P^T)$; `$'L'$`: Lower triangular, form is $A = P*L*D*(L^H)*(P^T)$.
- **n**: The order of the matrix $A$. $n \geq 0$.
- **A**: Array of size $\max(1, lda*n)$. Diagonal of the block diagonal matrix $D$ and factor $U$ or $L$ as computed by `?hetrf_rk`:
  - *Only* diagonal elements of the Hermitian block diagonal matrix $D$ on the diagonal of $A$—that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of $D$ must be provided on entry in array $e$.
  - and—
  - If $uplo = 'U'$, factor $U$ in the superdiagonal part of $A$. If $uplo = 'L'$, factor $L$ in the subdiagonal part of $A$.
- **lda**: The leading dimension of the array $A$.
- **e**: Array of size $n$. On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix $D$ with 1-by-1 or 2-by-2 diagonal blocks. If $uplo = 'U'$, $e(i) = D(i-1,i), i=2:N$, and $e(1)$ is not referenced. If $uplo = 'L'$, $e(i) = D(i+1,i), i=1:N-1$, and $e(n)$ is not referenced.

**NOTE** For 1-by-1 diagonal block $D(k)$, where $1 \leq k \leq n$, the element $e[k-1]$ is not referenced in both the $uplo = 'U'$ and $uplo = 'L'$ cases.

- **ipiv**: Array of size $n$. Details of the interchanges and the block structure of $D$ as determined by `?hetrf_rk`.
- **anorm**: The 1-norm of the original matrix $A$.

**Output Parameters**

- **rcond**: The reciprocal of the condition number of the matrix $A$, computed as $rcond = 1 / (\text{anorm} \times AINVNM)$, where $AINVNM$ is an estimate of the 1-norm of inv($A$) computed in this routine.

**Return Values**

This function returns a value $info$.

- $= 0$: Successful exit.
< 0: If info = -i, the i-th argument had an illegal value.

**?spcon**
Estimates the reciprocal of the condition number of a packed symmetric matrix.

**Syntax**
lapack_int LAPACKE_sspcon( int matrix_layout, char uplo, lapack_int n, const float* ap, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_dspcon( int matrix_layout, char uplo, lapack_int n, const double* ap, const lapack_int* ipiv, double anorm, double* rcond );
lapack_int LAPACKE_cspcon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );
lapack_int LAPACKE_zspcon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );

**Include Files**
- mkl.h

**Description**
The routine estimates the reciprocal of the condition number of a packed symmetric matrix $A$:

$$
\kappa_1(A) = \frac{||A||_1 ||A^{-1}||_1}{(||A||_1 ||A^{-1}||_1)} \quad \text{(since A is symmetric, } \kappa_\infty(A) = \kappa_1(A)).
$$

An estimate is obtained for $||A^{-1}||$, and the reciprocal of the condition number is computed as $rcond = 1 / (||A|| \cdot ||A^{-1}||)$.

Before calling this routine:
- compute $anorm$ (either $||A||_1 = \max_{i} \sum_{j} |a_{ij}|$ or $||A||_\infty = \max_{j} \sum_{i} |a_{ij}|$)
- call ?sptrf to compute the factorization of $A$.

**Input Parameters**
- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**
  Must be 'U' or 'L'.
  Indicates how the input matrix $A$ has been factored:
  - If $uplo = 'U'$, the array $ap$ stores the packed upper triangular factor $U$ of the factorization $A = U^*D*U^T$.
  - If $uplo = 'L'$, the array $ap$ stores the packed lower triangular factor $L$ of the factorization $A = L^*D*L^T$.
- **n**
  The order of matrix $A$; $n \geq 0$.
- **ap**
  The array $ap$ contains the packed factored matrix $A$, as returned by ?sptrf. The dimension of $ap$ must be at least $max(1,n(n+1)/2)$.
- **ipiv**
  Array, size at least $max(1, n)$.
  The array $ipiv$, as returned by ?sptrf.
- **anorm**
  The norm of the original matrix $A$ (see Description).
Output Parameters

\( rcond \)
An estimate of the reciprocal of the condition number. The routine sets \( rcond = 0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( rcond \) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The computed \( rcond \) is never less than \( r \) (the reciprocal of the true condition number) and in practice is nearly always less than 10\( r \). A call to this routine involves solving a number of systems of linear equations \( A^* x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors and \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

?hpcon
Estimates the reciprocal of the condition number of a packed Hermitian matrix.

Syntax

```c
lapack_int LAPACKE_chpcon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* ap, const lapack_int* ipiv, float anorm, float* rcond );
```

```c
lapack_int LAPACKE_zhpcon( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* ap, const lapack_int* ipiv, double anorm, double* rcond );
```

Include Files

- mkl.h

Description

The routine estimates the reciprocal of the condition number of a Hermitian matrix \( A \):

\[
\kappa_1(A) = \| A \|_1 / \| A^{-1} \|_1 \quad (\text{since } A \text{ is Hermitian}, \quad \kappa_\infty(A) = \kappa_1(A)).
\]

An estimate is obtained for \( \| A^{-1} \|_1 \), and the reciprocal of the condition number is computed as \( rcond = 1 / (\| A \|_1 \| A^{-1} \|) \).

Before calling this routine:

- compute \( \text{anorm} \) (either \( \| A \|_1 = \max_i \sum_j | a_{ij} | \) or \( \| A \|_\infty = \max_j \sum_i | a_{ij} | \))
- call ?hptrf to compute the factorization of \( A \).

Input Parameters

\( matrix_layout \)
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
uplo

Must be 'U' or 'L'.

Indicates how the input matrix $A$ has been factored:
If $\text{uplo} = 'U'$, the array $ap$ stores the packed upper triangular factor $U$ of the factorization $A = UDU^T$.
If $\text{uplo} = 'L'$, the array $ap$ stores the packed lower triangular factor $L$ of the factorization $A = LDL^T$.

$n$

The order of matrix $A$; $n \geq 0$.

$ap$

The array $ap$ contains the packed factored matrix $A$, as returned by $\text{?hptrf}$. The dimension of $ap$ must be at least $\max(1, n(n+1)/2)$.

$ipiv$

Array, size at least $\max(1, n)$. The array $ipiv$, as returned by $\text{?hptrf}$.

$anorm$

The norm of the original matrix $A$ (see Description).

Output Parameters

$rcond$

An estimate of the reciprocal of the condition number. The routine sets $rcond = 0$ if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime $rcond$ is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

Application Notes

The computed $rcond$ is never less than $r$ (the reciprocal of the true condition number) and in practice is nearly always less than $10r$. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 5 and never more than 11. Each solution requires approximately $8n^2$ floating-point operations.

See Also

Matrix Storage Schemes

$\text{?trcon}$

Estimates the reciprocal of the condition number of a triangular matrix.

Syntax

```
lapack_int LAPACKE_strcon ( int matrix_layout, char norm, char uplo, char diag,
    lapack_int n, const float* a, lapack_int lda, float* rcond );
lapack_int LAPACKE_dtrcon ( int matrix_layout, char norm, char uplo, char diag,
    lapack_int n, const double* a, lapack_int lda, double* rcond );
lapack_int LAPACKE_ctrcon ( int matrix_layout, char norm, char uplo, char diag,
    lapack_int n, const lapack_complex_float* a, lapack_int lda, float* rcond );
```
lapack_int LAPACKE_ztrcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, const lapack_complex_double* a, lapack_int lda, double* rcond );

Include Files
• mkl.h

Description
The routine estimates the reciprocal of the condition number of a triangular matrix A in either the 1-norm or infinity-norm:
k_1(A) = ||A||_1 ||A^{-1}||_1 = \kappa_1(A^T) = \kappa_1(A^H)
k_\infty(A) = ||A||_\infty ||A^{-1}||_\infty = \kappa_1(A^T) = \kappa_1(A^H).

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

norm
Must be '1' or 'O' or 'I'.
If norm = '1' or 'O', then the routine estimates the condition number of matrix A in 1-norm.
If norm = 'I', then the routine estimates the condition number of matrix A in infinity-norm.

uplo
Must be 'U' or 'L'.
Indicates whether A is upper or lower triangular:
If uplo = 'U', the array a stores the upper triangle of A, other array elements are not referenced.
If uplo = 'L', the array a stores the lower triangle of A, other array elements are not referenced.

diag
Must be 'N' or 'U'.
If diag = 'N', then A is not a unit triangular matrix.
If diag = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array a.

n
The order of the matrix A; n \geq 0.

a
The array a of size max(1, lda*n) contains the matrix A.

lda
The leading dimension of a; lda \geq max(1, n).

Output Parameters

rcond
An estimate of the reciprocal of the condition number. The routine sets rcond =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime rcond is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

Application Notes

The computed `rcond` is never less than `r` (the reciprocal of the true condition number) and in practice is nearly always less than `10r`. A call to this routine involves solving a number of systems of linear equations `A*x = b`; the number is usually 4 or 5 and never more than 11. Each solution requires approximately `n^2` floating-point operations for real flavors and `4n^2` operations for complex flavors.

See Also

Matrix Storage Schemes

`?tpcon`
Estimates the reciprocal of the condition number of a packed triangular matrix.

Syntax

```c
lapack_int LAPACKE_stpcon( int matrix_layout, char norm, char uplo, char diag,
                         lapack_int n, const float* ap, float* rcond );

lapack_int LAPACKE_dtpcon( int matrix_layout, char norm, char uplo, char diag,
                         lapack_int n, const double* ap, double* rcond );

lapack_int LAPACKE_ctpcon( int matrix_layout, char norm, char uplo, char diag,
                         lapack_int n, const lapack_complex_float* ap, float* rcond );

lapack_int LAPACKE_ztpcon( int matrix_layout, char norm, char uplo, char diag,
                         lapack_int n, const lapack_complex_double* ap, double* rcond );
```

Include Files

- `mkl.h`

Description

The routine estimates the reciprocal of the condition number of a packed triangular matrix `A` in either the 1-norm or infinity-norm:

\[
\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_e(A^T) = \kappa_e(A^H)
\]

\[
\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).
\]

Input Parameters

- `matrix_layout`
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- `norm`
  Must be '1' or 'O' or 'I'.
  - If `norm = '1'` or 'O', then the routine estimates the condition number of matrix `A` in 1-norm.
  - If `norm = 'I'`, then the routine estimates the condition number of matrix `A` in infinity-norm.
uplo

Must be 'U' or 'L'. Indicates whether A is upper or lower triangular:
If uplo = 'U', the array ap stores the upper triangle of A in packed form.
If uplo = 'L', the array ap stores the lower triangle of A in packed form.

diag

Must be 'N' or 'U'.
If diag = 'N', then A is not a unit triangular matrix.
If diag = 'U', then A is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array ap.

n

The order of the matrix A; n ≥ 0.

ap

The array ap contains the packed matrix A. The dimension of ap must be at least max(1, n(n+1)/2).

Output Parameters

rcond

An estimate of the reciprocal of the condition number. The routine sets rcond = 0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime rcond is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

Return Values

This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.

Application Notes

The computed rcond is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than 10r. A call to this routine involves solving a number of systems of linear equations A*x = b; the number is usually 4 or 5 and never more than 11. Each solution requires approximately n^2 floating-point operations for real flavors and 4n^2 operations for complex flavors.

See Also
Matrix Storage Schemes

?tbcon
Estimates the reciprocal of the condition number of a triangular band matrix.

Syntax

lapack_int LAPACKE_stbcon( int matrix_layout, char norm, char uplo, char diag, lapack_int n, lapack_int kd, const float* ab, lapack_int ldab, float* rcond );

lapack_int LAPACKE_dtbcon( int matrix_layout, char norm, char uplo, char diag, lapack_int n, lapack_int kd, const double* ab, lapack_int ldab, double* rcond );

lapack_int LAPACKE_ctbcon( int matrix_layout, char norm, char uplo, char diag, lapack_int n, lapack_int kd, const lapack_complex_float* ab, lapack_int ldab, float* rcond );
lapack_int LAPACKE_ztbcon( int matrix_layout, char norm, char uplo, char diag,
lapack_int n, lapack_int kd, const lapack_complex_double* ab, lapack_int ldab, double* rcond );

Include Files
• mkl.h

Description
The routine estimates the reciprocal of the condition number of a triangular band matrix \( A \) in either the 1-norm or infinity-norm:

\[
\kappa_1(A) = \|A\|_1 \|A^{-1}\|_1 = \kappa_\infty(A^T) = \kappa_\infty(A^H)
\]

\[
\kappa_\infty(A) = \|A\|_\infty \|A^{-1}\|_\infty = \kappa_1(A^T) = \kappa_1(A^H).
\]

Input Parameters

\textit{matrix_layout}  
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\textit{norm}  
Must be '1' or 'O' or 'I'.

If \textit{norm} = '1' or 'O', then the routine estimates the condition number of matrix \( A \) in 1-norm.

If \textit{norm} = 'I', then the routine estimates the condition number of matrix \( A \) in infinity-norm.

\textit{uplo}  
Must be 'U' or 'L'. Indicates whether \( A \) is upper or lower triangular:

If \textit{uplo} = 'U', the array \textit{ap} stores the upper triangle of \( A \) in packed form.

If \textit{uplo} = 'L', the array \textit{ap} stores the lower triangle of \( A \) in packed form.

\textit{diag}  
Must be 'N' or 'U'.

If \textit{diag} = 'N', then \( A \) is not a unit triangular matrix.

If \textit{diag} = 'U', then \( A \) is unit triangular: diagonal elements are assumed to be 1 and not referenced in the array \textit{ab}.

\textit{n}  
The order of the matrix \( A \); \( n \geq 0 \).

\textit{kd}  
The number of superdiagonals or subdiagonals in the matrix \( A \); \( kd \geq 0 \).

\textit{ab}  
The array \textit{ab} of size \( \max(1, ldab*\textit{n}) \) contains the band matrix \( A \).

\textit{ldab}  
The leading dimension of the array \textit{ab}. (\( ldab \geq kd +1 \)).

Output Parameters

\textit{rcond}  
An estimate of the reciprocal of the condition number. The routine sets \textit{rcond} =0 if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \textit{rcond} is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.
Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.

Application Notes
The computed rcond is never less than r (the reciprocal of the true condition number) and in practice is nearly always less than 10r. A call to this routine involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n(kd + 1)$ floating-point operations for real flavors and $8n(kd + 1)$ operations for complex flavors.

See Also
Matrix Storage Schemes

Refracting the Solution and Estimating Its Error: LAPACK Computational Routines
This section describes the LAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Routines for Solving Systems of Linear Equations).

?gerfs
Refrains the solution of a system of linear equations with a general coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sgerfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dgerfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_cgerfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zgerfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```
The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ or $A^T*X = B$ or $A^H*X = B$ with a general matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error $\beta$. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here $x_e$ is the exact solution).

Before calling this routine:
- call the factorization routine ?getrf
- call the solver routine ?getrs.

### Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **trans**
  - Must be 'N' or 'T' or 'C'.
  - Indicates the form of the equations:
    - If $trans = 'N'$, the system has the form $A*X = B$.
    - If $trans = 'T'$, the system has the form $A^T*X = B$.
    - If $trans = 'C'$, the system has the form $A^H*X = B$.

- **n**
  - The order of the matrix $A$; $n \geq 0$.

- **nrhs**
  - The number of right-hand sides; $nrhs \geq 0$.

- **a, af, b, x**
  - Arrays:
    - $a(\text{size max}(1, \text{lda} * n))$ contains the original matrix $A$, as supplied to ?getrf.
    - $af(\text{size max}(1, \text{ldaf} * n))$ contains the factored matrix $A$, as returned by ?getrf.
    - $b(\text{size max}(1, \text{ldb} * nrhs))$ for column major layout and max$(1, \text{ldb} * n)$ for row major layout contains the right-hand side matrix $B$.
    - $x(\text{size max}(1, \text{ldx} * nrhs))$ for column major layout and max$(1, \text{ldx} * n)$ for row major layout contains the solution matrix $X$.

- **lda**
  - The leading dimension of $a$; $lda \geq \text{max}(1, n)$.

- **ldaf**
  - The leading dimension of $af$; $ldaf \geq \text{max}(1, n)$.

- **ldb**
  - The leading dimension of $b$; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq \text{nrhs}$ for row major layout.

- **ldx**
  - The leading dimension of $x$; $ldx \geq \text{max}(1, n)$ for column major layout and $ldx \geq \text{nrhs}$ for row major layout.

- **ipiv**
  - Array, size at least $\text{max}(1, n)$.
  - The ipiv array, as returned by ?getrf.
Output Parameters

- x: The refined solution matrix X.
- ferr, berr: Arrays, size at least \( \max(1, nrhs) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, parameter i had an illegal value.

Application Notes

The bounds returned in ferr are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of \( 4n^2 \) floating-point operations (for real flavors) or \( 16n^2 \) operations (for complex flavors). In addition, each step of iterative refinement involves \( 6n^2 \) operations (for real flavors) or \( 24n^2 \) operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations \( Ax = b \) with the same coefficient matrix \( A \) and different right hand sides \( b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors or \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

?gerfsx
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a general coefficient matrix \( A \) and provides error bounds and backward error estimates.

Syntax

```c
lapack_int LAPACKE_sgerfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int* ipiv, const float* r, const float* c, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_dgerfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf, const lapack_int* ipiv, const double* r, const double* c, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_cgerfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* r, const float* c, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );
```
LAPACKE_zgerfsx:

```c
int LAPACKE_zgerfsx( int matrix_layout, char trans, char equed, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* r,
const double* c, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double*
x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double*
err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );
```

### Include Files
- mkl.h

### Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed`, `r`, and `c` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

### Input Parameters

- **matrix_layout**
  Specifies whether two-dimensional array storage is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **trans**
  Must be 'N', 'T', or 'C'.

  Specifies the form of the system of equations:
  - If `trans` = 'N', the system has the form $A * X = B$ (No transpose).
  - If `trans` = 'T', the system has the form $A^T * X = B$ (Transpose).
  - If `trans` = 'C', the system has the form $A^H * X = B$ (Conjugate transpose for complex flavors, Transpose for real flavors).

- **equed**
  Must be 'N', 'R', 'C', or 'B'.

  Specifies the form of equilibration that was done to $A$ before calling this routine.
  - If `equed` = 'N', no equilibration was done.
  - If `equed` = 'R', row equilibration was done, that is, $A$ has been premultiplied by $\text{diag}(r)$.
  - If `equed` = 'C', column equilibration was done, that is, $A$ has been postmultiplied by $\text{diag}(c)$.
  - If `equed` = 'B', both row and column equilibration was done, that is, $A$ has been replaced by $\text{diag}(r) * A * \text{diag}(c)$. The right-hand side $B$ has been changed accordingly.

- **n**
  The number of linear equations; the order of the matrix $A$; $n \geq 0$.

- **nrhs**
  The number of right-hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$. 

Arrays: $a$ (size max(1, \(lda\times n\))), $af$ (size max(1, \(ldaf\times n\))), $b$ (size max(1, \(ldb\times nrhs\))) for column major layout and max(1, \(ldb\times n\)) for row major layout).

The array $a$ contains the original $n$-by-$n$ matrix $A$.

The array $af$ contains the factored form of the matrix $A$, that is, the factors $L$ and $U$ from the factorization $A = P*L*U$ as computed by \texttt{getrf}.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

The leading dimension of $a$; \(lda\geq\max(1, n)\).

The leading dimension of $af$; \(ldaf\geq\max(1, n)\).

Array, size at least max(1, \(n\)). Contains the pivot indices as computed by \texttt{getrf}; for row \(1 \leq i \leq n\), row $i$ of the matrix was interchanged with row $ipiv(i)$.

Arrays: $r$ (size $n$), $c$ (size $n$). The array $r$ contains the row scale factors for $A$, and the array $c$ contains the column scale factors for $A$.

If $equed = 'R'$ or 'B', $A$ is multiplied on the left by $\text{diag}(r)$; if $equed = 'N'$ or 'C', $r$ is not accessed.

If $equed = 'R'$ or 'B', each element of $r$ must be positive.

If $equed = 'C'$ or 'B', $A$ is multiplied on the right by $\text{diag}(c)$; if $equed = 'N'$ or 'R', $c$ is not accessed.

If $equed = 'C'$ or 'B', each element of $c$ must be positive.

Each element of $r$ or $c$ should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array $b$; \(ldb\geq\max(1, n)\) for column major layout and \(ldb\geq\text{nrhs}\) for row major layout.

Array, of size max(1, \(ldx\times nrhs\))) for column major layout and max(1, \(ldx\times n\)) for row major layout.

The solution matrix $X$ as computed by \texttt{getrs}.

The leading dimension of the output array $x$; \(ldx\geq\max(1, n)\) for column major layout and \(ldx\geq\text{nrhs}\) for row major layout.

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See \texttt{err_bnds_norm} and \texttt{err_bnds_comp} descriptions in Output Arguments section below.

Specifies the number of parameters set in \texttt{params}. If \(\leq 0\), the \texttt{params} array is never referenced and default values are used.
Array, size nparams. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to nparams are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass nparams = 0, which prevents the source code from accessing the params argument.

params[0] : Whether to perform iterative refinement or not. Default: 1.0

=0.0  No refinement is performed and no error bounds are computed.

=1.0  Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

params[1] : Maximum number of residual computations allowed for refinement.

Default  10.0

Aggressive  Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in err_bnds_norm and err_bnds_comp may no longer be trustworthy.

params[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

Output Parameters

x  The improved solution matrix X.

rcond  Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If rcond is less than the machine precision, in particular, if rcond = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

berr  Array, size at least max(1, nrhs). Contains the componentwise relative backward error for each solution vector x_j, that is, the smallest relative change in any element of A or B that makes x_j an exact solution.
Array of size \( nrhs * n_{\text{err_bnds}} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the \( i \)-th solution vector

\[
\frac{\max_j |X_{true,j} - X_{j,i}|}{\max_j |X_{j,i}|}
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

**err=1**  "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} * \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} * \text{dlamch}(\varepsilon) \) for double precision flavors.

**err=2**  "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} * \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} * \text{dlamch}(\varepsilon) \) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

**err=3**  Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold \( \sqrt{n} * \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} * \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \( Z \) are:

\[
\|Z\|_\infty \cdot \|Z^{-1}\|_\infty
\]

Let \( z=s*a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq nrhs \), and type of error \( err \) is stored in:

- **Column major layout:** \( \text{err_bnds_norm}[(err - 1)*nrhs + i - 1] \).
- **Row major layout:** \( \text{err_bnds_norm}[err - 1 + (i - 1)*n_{\text{err_bnds}}] \)
err_bnds_comp

Array of size nrhs\times n_{\text{err\_bnds}}. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \(i\)-th solution vector:

\[
\max_j \frac{|X_{\text{true},ji} - X_{ji}|}{|X_{ji}|}
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (\(\text{params}[2] = 0.0\)), then \(\text{err\_bnds\_comp}\) is not accessed. If \(n_{\text{err\_bnds}} < 3\), then at most the first \(n_{\text{err\_bnds}}\) columns of the \(\text{err\_bnds\_comp}\) array are returned.

**err=1**

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \(\sqrt{n} \cdot \text{slamch(\varepsilon)}\) for single precision flavors and \(\sqrt{n} \cdot \text{dlamch(\varepsilon)}\) for double precision flavors.

**err=2**

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \(\sqrt{n} \cdot \text{slamch(\varepsilon)}\) for single precision flavors and \(\sqrt{n} \cdot \text{dlamch(\varepsilon)}\) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

**err=3**

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold \(\sqrt{n} \cdot \text{slamch(\varepsilon)}\) for single precision flavors and \(\sqrt{n} \cdot \text{dlamch(\varepsilon)}\) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \(Z\) are:

\[
\|Z\|_\infty \cdot \|Z^{-1}\|_\infty
\]

Let \(z = s \cdot (a \cdot \text{diag}(x))\), where \(x\) is the solution for the current right-hand side and \(s\) scales each row of \(a \cdot \text{diag}(x)\) by a power of the radix so all absolute row sums of \(z\) are approximately 1.

The information for right-hand side \(i\), where \(1 \leq i \leq \text{nrhs}\), and type of error \(\text{err}\) is stored in:
• Column major layout: \( \text{err}_\text{bnds}_\text{comp}[(\text{err} - 1) \times \text{nrhs} + i - 1] \).
• Row major layout: \( \text{err}_\text{bnds}_\text{comp}[\text{err} - 1 + (i - 1) \times \text{n}_\text{err}_\text{bnds}] \)

params

Output parameter only if the input contains erroneous values, namely, in \( \text{params}[0], \text{params}[1], \text{params}[2] \). In such a case, the corresponding elements of params are filled with default values on output.

Return Values

This function returns a value info.

If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( 0 < \text{info} < \text{n} \): \( \text{U}_\text{info} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( r\text{cond} = 0 \) is returned.

If \( \text{info} = \text{n} + j \): The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest \( j \) such that for column major layout \( \text{err}_\text{bnds}_\text{norm}[j - 1] = 0.0 \) or \( \text{err}_\text{bnds}_\text{comp}[j - 1] = 0.0 \); or for row major layout \( \text{err}_\text{bnds}_\text{norm}[(j - 1) \times \text{n}_\text{err}_\text{bnds}] = 0.0 \) or \( \text{err}_\text{bnds}_\text{comp}[(j - 1) \times \text{n}_\text{err}_\text{bnds}] = 0.0 \)). See the definition of \( \text{err}_\text{bnds}_\text{norm} \) and \( \text{err}_\text{bnds}_\text{comp} \) for \( \text{err} = 1 \). To get information about all of the right-hand sides, check \( \text{err}_\text{bnds}_\text{norm} \) or \( \text{err}_\text{bnds}_\text{comp} \).

See Also

Matrix Storage Schemes

?gbrfs

Refines the solution of a system of linear equations with a general band coefficient matrix and estimates its error.

Syntax

```
lapack_int LAPACKE_sgbtrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb, lapack_int ldafb, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dgbtrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb, lapack_int ldafb, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cgbtrfs( int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab, const lapack_complex_float* afb, lapack_int ldafb, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```
lapack_int LAPACKE_zgbrfs ( int matrix_layout, char trans, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab, const lapack_complex_double* afb, lapack_int ldafb, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );

Include Files

• mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ or $A^T \cdot X = B$ with a band matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error $\beta$. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the component-wise forward error in the computed solution $\|x - x_\|\|_\|_\infty / \|x_\|_\|_\infty$ (here $x_\|$ is the exact solution).

Before calling this routine:

• call the factorization routine ?gbtrf
• call the solver routine ?gbtrs.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

trans

Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If trans = 'N', the system has the form $A \cdot X = B$.

If trans = 'T', the system has the form $A^T \cdot X = B$.

If trans = 'C', the system has the form $A^H \cdot X = B$.

n

The order of the matrix $A$; $n \geq 0$.

kl

The number of sub-diagonals within the band of $A$; $kl \geq 0$.

ku

The number of super-diagonals within the band of $A$; $ku \geq 0$.

nrhs

The number of right-hand sides; $nrhs \geq 0$.

ab, afb, b, x

Arrays:

$ab$(size max(1, ldab*n)) contains the original band matrix $A$, as supplied to ?gbtrf, but stored in rows from 1 to $kl + ku + 1$ for column major layout, and columns from 1 to $kl + ku + 1$ for row major layout.

$afb$(size max(1, ldafb*n)) contains the factored band matrix $A$, as returned by ?gbtrf.

$b$(size max(1, ldb*nrhs) for column major layout and max(1, ldx*n) for row major layout) contains the right-hand side matrix $B$. 
The leading dimension of $ab$, $ldab \geq kl + ku + 1$.

The leading dimension of $afb$, $ldafb \geq 2*kl + ku + 1$.

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

The leading dimension of $x$; $ldx \geq \max(1, n)$.

Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by $?gbtrf$.

### Output Parameters

- $x$:
The refined solution matrix $X$.

- $ferr, berr$:
Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

### Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

### Application Notes

The bounds returned in $ferr$ are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n(kl + ku)$ floating-point operations (for real flavors) or $16n(kl + ku)$ operations (for complex flavors). In addition, each step of iterative refinement involves $2n(4kl + 3ku)$ operations (for real flavors) or $8n(4kl + 3ku)$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

### See Also

**Matrix Storage Schemes**

$?gbfsx$

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a banded coefficient matrix $A$ and provides error bounds and backward error estimates.

### Syntax

```c
lapack_int LAPACKE_sgbrfsx( int matrix_layout, char trans, char equed, lapack_int n,
    lapack_int kl, lapack_int ku, lapack_int nrhs, const float* ab, lapack_int ldab,
    const float* afb, lapack_int ldafb, const lapack_int* ipiv, const float* r,
    const float* c, const float* b, lapack_int ldb, float* x, lapack_int ldx,
    float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm,
    float* err_bnds_comp, lapack_int nparams, float* params );
```
lapack_int LAPACK_dgbrfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb, lapack_int ldafb, const lapack_int* ipiv, const double* r, const double* c, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACK_cgbrfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab, const lapack_complex_float* afb, lapack_int ldafb, const lapack_int* ipiv, const float* r, const float* c, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACK_zgbrfsx( int matrix_layout, char trans, char equed, lapack_int n, lapack_int kl, lapack_int ku, lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab, const lapack_complex_double* afb, lapack_int ldafb, const lapack_int* ipiv, const double* r, const double* c, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for err_bnds_norm and err_bnds_comp for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters equed, r, and c below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

matrix_layout

Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

trans

Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

- If trans = 'N', the system has the form A*X = B (No transpose).
- If trans = 'T', the system has the form A^T*X = B (Transpose).
- If trans = 'C', the system has the form A^H*X = B (Conjugate transpose for complex flavors, Transpose for real flavors).

equed

Must be 'N', 'R', 'C', or 'B'.

Specifies the form of equilibration that was done to A before calling this routine.

- If equed = 'N', no equilibration was done.
If `equed = 'R'`, row equilibration was done, that is, \( A \) has been premultiplied by \( \text{diag}(r) \).

If `equed = 'C'`, column equilibration was done, that is, \( A \) has been postmultiplied by \( \text{diag}(c) \).

If `equed = 'B'`, both row and column equilibration was done, that is, \( A \) has been replaced by \( \text{diag}(r) \cdot A \cdot \text{diag}(c) \). The right-hand side \( B \) has been changed accordingly.

The number of linear equations; the order of the matrix \( A \); \( n \geq 0 \).

The number of subdiagonals within the band of \( A \); \( kl \geq 0 \).

The number of superdiagonals within the band of \( A \); \( ku \geq 0 \).

The number of right-hand sides; the number of columns of the matrices \( B \) and \( X \); \( nrhs \geq 0 \).

The array \( ab \) of size \( \max(1, \text{ldab} \cdot n) \) contains the original matrix \( A \) in band storage, in rows from 1 to \( kl + ku + 1 \) for column major layout, and in columns from 1 to \( kl + ku + 1 \) for row major layout.

The array \( afb \) of size \( \max(1, \text{ldafb} \cdot n) \) contains details of the LU factorization of the banded matrix \( A \) as computed by \( ?\text{gbtrf} \).

The array \( b \) of size \( \max(1, \text{ldb} \cdot nrhs) \) for column major layout and \( \max(1, \text{ldb} \cdot n) \) for row major layout contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

The leading dimension of the array \( ab \); \( \text{ldab} \geq kl + ku + 1 \).

The leading dimension of the array \( afb \); \( \text{ldafb} \geq 2 \cdot kl + ku + 1 \).

Array, size at least \( \max(1, n) \). Contains the pivot indices as computed by \( ?\text{gbtrf} \); for row \( 1 \leq i \leq n \), row \( i \) of the matrix was interchanged with row \( \text{ipiv}[i-1] \).

Arrays: \( r(n) \), \( c(n) \). The array \( r \) contains the row scale factors for \( A \), and the array \( c \) contains the column scale factors for \( A \).

If `equed = 'R'` or `'B'`, \( A \) is multiplied on the left by \( \text{diag}(r) \); if `equed = 'N'` or `'C'`, \( r \) is not accessed.

If `equed = 'R'` or `'B'`, each element of \( r \) must be positive.

If `equed = 'C'` or `'B'`, \( A \) is multiplied on the right by \( \text{diag}(c) \); if `equed = 'N'` or `'R'`, \( c \) is not accessed.

If `equed = 'C'` or `'B'`, each element of \( c \) must be positive.

Each element of \( r \) or \( c \) should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array \( b \); \( \text{ldb} \geq \max(1, n) \) for column major layout and \( \text{ldb} \geq nrhs \) for row major layout.
x

Array, size max(1, ldx*nrhs) for column major layout and max(1, ldx*n) for row major layout.
The solution matrix $X$ as computed by `sbgtrs/dgbtrs` for real flavors or `cgbtrs/zgbtrs` for complex flavors.

ldx

The leading dimension of the output array $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

n_err_bnds

Number of error bounds to return for each right-hand side and each type (normwise or componentwise). See `err_bnds_norm` and `err_bnds_comp` descriptions in **Output Arguments** section below.

nparams

Specifies the number of parameters set in `params`. If $\leq 0$, the `params` array is never referenced and default values are used.

params

Array, size `nparams`. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to `nparams` are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass `nparams = 0`, which prevents the source code from accessing the `params` argument.

`params[0]`: Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

- **=0.0**: No refinement is performed and no error bounds are computed.
- **=1.0**: Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

`params[1]`: Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive

Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]`: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

**Output Parameters**

x

The improved solution matrix $X$. 

---

536
Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

Array, size at least max(1, nrhs). Contains the componentwise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

Array of size $nrhs \times n\_err\_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

$$\frac{\max_j |X_{true,j} - X_{j,i}|}{\max_j |X_{j,i}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

$err=1$  
"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors.

$err=2$  
"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

$err=3$  
Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let $z = s \times a$, where $s$ scales each row by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq irhs$, and type of error $err$ is stored in:
- Column major layout: $\text{err\_bnds\_norm}[(\text{err} - 1)\cdot \text{nrhs} + i - 1]$.
- Row major layout: $\text{err\_bnds\_norm}[	ext{err} - 1 + (i - 1)\cdot n\_err\_bnds]$

Array of size $\text{nrhs}\cdot n\_err\_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the $i$-th solution vector:

$$\max \frac{|X_{true_{ij}} - X_{ii}|}{|X_{ii}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}[2] = 0.0$), then $\text{err\_bnds\_comp}$ is not accessed.

$\text{err}=1$  
"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n}\cdot \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n}\cdot \text{dlamch}(\varepsilon)$ for double precision flavors.

$\text{err}=2$  
"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n}\cdot \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n}\cdot \text{dlamch}(\varepsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

$\text{err}=3$  
Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n}\cdot \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n}\cdot \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are

$$\|Z\|\cdot \|Z^{-1}\|$$

Let $z = s\cdot(a\cdot\text{diag}(x))$, where $x$ is the solution for the current right-hand side and $s$ scales each row of $a\cdot\text{diag}(x)$ by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error $\text{err}$ is stored in:

- Column major layout: $\text{err\_bnds\_comp}[(\text{err} - 1)\cdot \text{nrhs} + i - 1]$.
Row major layout: \( \text{err\_bnds\_comp}[\text{err} - 1 + (i - 1)\times\text{n\_err\_bnds}] \)

Output parameter only if the input contains erroneous values, namely, in \( \text{params}[0], \text{params}[1], \) and \( \text{params}[2] \). In such a case, the corresponding elements of \( \text{params} \) are filled with default values on output.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( 0 < \text{info} \leq n \): \( \text{U}^{\text{info}} \), \( \text{info} \) is exactly zero. The factorization has been completed, but the factor \( \text{U} \) is exactly singular, so the solution and error bounds could not be computed; \( \text{rcond} = 0 \) is returned.

If \( \text{info} = n + j \): The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest \( j \) such that for column major layout \( \text{err\_bnds\_norm}[j - 1] = 0.0 \) or \( \text{err\_bnds\_comp}[j - 1] = 0.0 \); or for row major layout \( \text{err\_bnds\_norm}[(j - 1)\times\text{n\_err\_bnds}] = 0.0 \) or \( \text{err\_bnds\_comp}[(j - 1)\times\text{n\_err\_bnds}] = 0.0 \). See the definition of \( \text{err\_bnds\_norm} \) and \( \text{err\_bnds\_comp} \) for \( \text{err} = 1 \). To get information about all of the right-hand sides, check \( \text{err\_bnds\_norm} \) or \( \text{err\_bnds\_comp} \).

**See Also**

**Matrix Storage Schemes**

\( ?\text{gtrfs} \)

Refines the solution of a system of linear equations with a tridiagonal coefficient matrix and estimates its error.

**Syntax**

- For real matrices:
  ```c
  lapack_int LAPACKE_sgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const float* dl, const float* d, const float* du, const float* dlf, const float* df, const float* duf, const float* du2, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
  ```

- For double matrices:
  ```c
  lapack_int LAPACKE_dgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const double* dl, const double* d, const double* du, const double* dlf, const double* df, const double* duf, const double* du2, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
  ```

- For complex single precision matrices:
  ```c
  lapack_int LAPACKE_cgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_float* dl, const lapack_complex_float* d, const lapack_complex_float* du, const lapack_complex_float* dlf, const lapack_complex_float* df, const lapack_complex_float* duf, const lapack_complex_float* du2, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
  ```

- For complex double precision matrices:
  ```c
  lapack_int LAPACKE_zgtrfs( int matrix_layout, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double* dl, const lapack_complex_double* d, const lapack_complex_double* du, const lapack_complex_double* dlf, const lapack_complex_double* df, const lapack_complex_double* duf, const lapack_complex_double* du2, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
  ```
lapack_complex_double* df, const lapack_complex_double* duf, const
lapack_complex_double* du2, const lapack_int* ipiv, const lapack_complex_double* b,
lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );

Include Files
• mkl.h

Description
The routine performs an iterative refinement of the solution to a system of linear equations \( A \mathbf{x} = \mathbf{b} \) or \( A^\top \mathbf{x} = \mathbf{b} \) or \( A^H \mathbf{x} = \mathbf{b} \) with a tridiagonal matrix \( A \), with multiple right-hand sides. For each computed solution vector \( \mathbf{x} \), the routine computes the component-wise backward error \( \beta \). This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( \mathbf{x} \) is the exact solution of the perturbed system:

\[
\frac{|\delta a_{ij}|}{|a_{ij}|} \leq \beta, \quad \frac{|\delta b_i|}{|b_i|} \leq \beta \quad \text{such that} \quad (A + \delta A) \mathbf{x} = (b + \delta b).
\]

Finally, the routine estimates the component-wise forward error in the computed solution \( \| \mathbf{x} - \mathbf{x}_e \|_\infty / \| \mathbf{x} \|_\infty \) (here \( \mathbf{x}_e \) is the exact solution).

Before calling this routine:
• call the factorization routine ?gttrf
• call the solver routine ?gttrs.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **trans**: Must be 'N' or 'T' or 'C'.

- **n**: The order of the matrix \( A \); \( n \geq 0 \).

- **nrhs**: The number of right-hand sides, that is, the number of columns of the matrix \( B \); \( nrhs \geq 0 \).

- **dl**: Array \( d_l \) of size \( n - 1 \) contains the subdiagonal elements of \( A \).

- **d**: Array \( d \) of size \( n \) contains the diagonal elements of \( A \).

- **du**: Array \( d_u \) of size \( n - 1 \) contains the superdiagonal elements of \( A \).

- **dlf**: Array \( d_{lf} \) of size \( n - 1 \) contains the \((n - 1)\) multipliers that define the matrix \( L \) from the LU factorization of \( A \) as computed by ?gttrf.

- **df**: Array \( df \) of size \( n \) contains the \( n \) diagonal elements of the upper triangular matrix \( U \) from the LU factorization of \( A \).

- **duf**: Array \( d_{uf} \) of size \( n - 1 \) contains the \((n - 1)\) elements of the first superdiagonal of \( U \).

- **du2**: Array \( d_{u2} \) of size \( n - 2 \) contains the \((n - 2)\) elements of the second superdiagonal of \( U \).
$b$ (size max(1, $ldb \times nrhs$) for column major layout and max(1, $ldb \times n$) for row major layout) contains the right-hand side matrix $B$.

$x$ (size max(1, $ldx \times nrhs$) for column major layout and max(1, $ldx \times n$) contains the solution matrix $X$, as computed by $\texttt{?gttrs}$.

$ldb$ The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq \max(1, nrhs)$ for row major layout.

$ldx$ The leading dimension of $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq \max(1, nrhs)$ for row major layout.

$ipiv$ Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by $\texttt{?gttrf}$.

**Output Parameters**

$x$ The refined solution matrix $X$.

$ferr, berr$ Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

**See Also**

Matrix Storage Schemes

$\texttt{?porfs}$

*Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.*

**Syntax**

```c
lapack_int LAPACKE_sporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* a, lapack_int lda, const float* af, lapack_int ldaf, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* a, lapack_int lda, const double* af, lapack_int ldaf, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_cporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
lapack_int ldaf, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* 
x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zporfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af,
lapack_int ldaf, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* 
x, lapack_int ldx, double* ferr, double* berr );
```
Include Files
• mkl.h

Description
The routine performs an iterative refinement of the solution to a system of linear equations \( A \cdot X = B \) with a symmetric (Hermitian) positive definite matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( x \) is the exact solution of the perturbed system:

\[
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A) \cdot x = (b + \delta b).
\]

Finally, the routine estimates the component-wise forward error in the computed solution \( \|x - x_e\|_\infty / \|x\|_\infty \) (here \( x_e \) is the exact solution).

Before calling this routine:
• call the factorization routine \( ?potrf \)
• call the solver routine \( ?potrs \).

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'.
  - If \( \text{uplo} = 'U' \), the upper triangle of \( A \) is stored.
  - If \( \text{uplo} = 'L' \), the lower triangle of \( A \) is stored.
- **n**: The order of the matrix \( A ; n \geq 0 \).
- **nrhs**: The number of right-hand sides; \( nrhs \geq 0 \).
- **a**: Array (size max(1, lda*n)) contains the original matrix \( A \), as supplied to \( ?potrf \).
- **af**: Array (size max(1, ldaf*n)) contains the factored matrix \( A \), as returned by \( ?potrf \).
- **b**: Array (size max(1, ldb*nrhs)) for column major layout and max(1, ldb*n) for row major layout contains the right-hand side matrix \( B \).
  - The second dimension of \( B \) must be at least max(1, nrhs).
- **x**: Array (size max(1, ldx*nrhs)) for column major layout and max(1, ldx*n) for row major layout contains the solution matrix \( X \).
- **lda**: The leading dimension of \( a ; lda \geq \text{max}(1, n) \).
- **ldaf**: The leading dimension of \( af ; ldaf \geq \text{max}(1, n) \).
- **ldb**: The leading dimension of \( b ; ldb \geq \text{max}(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.
- **ldx**: The leading dimension of \( x ; ldx \geq \text{max}(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.
Output Parameters

\[ x \]

The refined solution matrix \( X \).

\[ ferr, berr \]

Arrays, size at least \( \max(1, nrhs) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The bounds returned in \( ferr \) are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of \( 4n^2 \) floating-point operations (for real flavors) or \( 16n^2 \) operations (for complex flavors). In addition, each step of iterative refinement involves \( 6n^2 \) operations (for real flavors) or \( 24n^2 \) operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations \( A^*x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors or \( 8n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

?porfsx

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric/Hermitian positive-definite coefficient matrix \( A \) and provides error bounds and backward error estimates.

Syntax

```c
lapack_int LAPACKE_sporfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const float* s, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_dporfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf, const double* s, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_cporfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af, lapack_int ldaf, const float* s, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );
```
lapack_int LAPACKE_zporfsx( int matrix_layout, char uplo, char equed, lapack_int n, 
apack_int nrhs, const lapack_complex_double* a, lapack_int lda, const 
apack_complex_double* af, lapack_int ldaf, const double* s, const 
apack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, 
double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* 
err_bnds_comp, lapack_int nparams, double* params );

Include Files
• mkl.h

Description
The routine improves the computed solution to a system of linear equations and provides error bounds and
backward error estimates for the solution. In addition to a normwise error bound, the code provides a
maximum componentwise error bound, if possible. See comments for err_bnds_norm and err_bnds_comp
for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described
by the parameters equed and s below. In this case, the solution and error bounds returned are for the
original unequilibrated system.

Input Parameters

**matrix_layout**
- Specifies whether two-dimensional array storage is row major
  (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**uplo**
- Must be 'U' or 'L'.
- Indicates whether the upper or lower triangular part of A is stored:
  - If uplo = 'U', the upper triangle of A is stored.
  - If uplo = 'L', the lower triangle of A is stored.

**equed**
- Must be 'N' or 'Y'.
- Specifies the form of equilibration that was done to A before calling this
  routine.
  - If equed = 'N', no equilibration was done.
  - If equed = 'Y', both row and column equilibration was done, that is, A has
    been replaced by diag(s)*A*diag(s). The right-hand side B has been
    changed accordingly.

**n**
- The number of linear equations; the order of the matrix A; n ≥ 0.

**nrhs**
- The number of right-hand sides; the number of columns of the matrices B
  and X; nrhs ≥ 0.

**a**
- The array a (size max(1, lda*n)) contains the symmetric/Hermitian matrix
  A as specified by uplo. If uplo = 'U', the leading n-by-n upper triangular
  part of a contains the upper triangular part of the matrix A and the strictly
  lower triangular part of a is not referenced. If uplo = 'L', the leading n-
  by-n lower triangular part of a contains the lower triangular part of the
  matrix A and the strictly upper triangular part of a is not referenced.
The array \( af \) (size max(1, \( ldaf \times n \))) contains the triangular factor \( L \) or \( U \) from the Cholesky factorization \( A = U^T U \) or \( A = L^T L \) as computed by \texttt{spotrf} for real flavors or \texttt{dpotrf} for complex flavors.

The array \( b \) (size max(1, \( ldb \times nrhs \)) for column major layout and max(1, \( ldb \times n \)) for row major layout) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

The leading dimension of \( a \); \( lda \geq \max(1, n) \).

The leading dimension of \( af \); \( ldaf \geq \max(1, n) \).

Array of size \( n \). The array \( s \) contains the scale factors for \( A \).

If \( \texttt{equed} = 'N' \), \( s \) is not accessed.

If \( \texttt{equed} = 'Y' \), each element of \( s \) must be positive.

Each element of \( s \) should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Array, size max(1, \( ldx \times nrhs \)) for column major layout and max(1, \( ldx \times n \)) for row major layout.

The solution matrix \( X \) as computed by \texttt{?potrs}

The leading dimension of the output array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See \texttt{err_bnds_norm} and \texttt{err_bnds_comp} descriptions in \texttt{Output Arguments} section below.

Specifies the number of parameters set in \( \texttt{params} \). If \( \leq 0 \), the \( \texttt{params} \) array is never referenced and default values are used.

Array, size \( nparams \). Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to \( nparams \) are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass \( nparams = 0 \), which prevents the source code from accessing the \( \texttt{params} \) argument.

\( \texttt{params}[0] \): Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D0 (for double precision flavors).

\( =0.0 \) No refinement is performed and no error bounds are computed.

\( =1.0 \) Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)
**params[1]**: Maximum number of residual computations allowed for refinement.

- **Default**: 10.0
- **Aggressive**: Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in **err_bnds_norm** and **err_bnds_comp** may no longer be trustworthy.

**params[2]**: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

### Output Parameters

- **x**: The improved solution matrix $X$.
- **rcond**: Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.
- **berr**: Array, size at least $\max(1, nrhs)$. Contains the componentwise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.
- **err_bnds_norm**: Array of size $nrhs*\text{n_err_bnds}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

\[
\frac{\max_j |x_j^{\text{true}} - x_j^i|}{\max_j |x_j^i|}
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

- **err=1**: "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} * \text{dlamch}(\epsilon)$ for double precision flavors.

- **err=2**: "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} * \text{slamch}(\epsilon)$ for single precision flavors and
Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \( Z \) are

\[
\|Z\|_\infty \cdot \|Z^{-1}\|_\infty
\]

Let \( z = s \cdot a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( err \) is stored in:

- Column major layout: \( \text{err_bnds_norm}[\text{(err - 1)} \cdot \text{nrhs} + i - 1] \).
- Row major layout: \( \text{err_bnds_norm}[(\text{err} - 1) + (i - 1) \cdot \text{n_err_bnds}] \)

Array of size \( \text{nrhs} \cdot \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \( i \)-th solution vector:

\[
\max_j \left\| \frac{X_{\text{true},ij} - X_{ij}}{|X_{ij}|} \right\|
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (\( \text{params}[2] = 0.0 \)), then \( \text{err_bnds_comp} \) is not accessed.

\( err = 1 \)

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors.

\( err = 2 \)

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and
sqrt(n)*dlamch(e) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold
sqrt(n)*slamch(e) for single precision flavors and sqrt(n)*dlamch(e) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are

\[ \|Z\|_\infty \cdot \|Z^{-1}\|_\infty \]

Let \( z = s \cdot (a \cdot \text{diag}(x)) \), where \( x \) is the solution for the current right-hand side and \( s \) scales each row of \( a \cdot \text{diag}(x) \) by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in:

- Column major layout: \( \text{err_bnds_comp}[(\text{err} - 1)\cdot\text{nrhs} + i - 1] \).
- Row major layout: \( \text{err_bnds_comp}[\text{err} - 1 + (i - 1)\cdot\text{n_err_bnds}] \).

Output parameter only if the input contains erroneous values, namely in \( \text{params}[0], \text{params}[1], \) or \( \text{params}[2] \). In such a case, the corresponding elements of \( \text{params} \) are filled with default values on output.

Return Values
This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( 0 < \text{info} < n \): \( U^{\text{info},\text{info}} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( rcond = 0 \) is returned.

If \( \text{info} = n+j \): The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest \( j \) such that for column major layout \( \text{err_bnds_norm}[j - 1] = 0.0 \) or \( \text{err_bnds_comp}[j - 1] = 0.0 \); or for row major layout \( \text{err_bnds_norm}[(j - 1)\cdot\text{n_err_bnds}] = 0.0 \) or \( \text{err_bnds_comp}[(j - 1)\cdot\text{n_err_bnds}] = 0.0 \)). See the definition of \( \text{err_bnds_norm} \) and \( \text{err_bnds_comp} \) for \( \text{err} = 1 \). To get information about all of the right-hand sides, check \( \text{err_bnds_norm} \) or \( \text{err_bnds_comp} \).

See Also
Matrix Storage Schemes
?pprfs
Refines the solution of a system of linear equations
with a symmetric (Hermitian) positive-definite
coefficient matrix stored in a packed format and
estimates its error.

Syntax

```c
lapack_int LAPACKE_spprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const float* ap, const float* afp, const float* b, lapack_int ldb, float* x, lapack_int
ldx, float* ferr, float* berr );
```

```c
lapack_int LAPACKE_dpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const double* ap, const double* afp, const double* b, lapack_int ldb, double* x,
lapack_int ldx, double* ferr, double* berr );
```

```c
lapack_int LAPACKE_cpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_float* ap, const lapack_complex_float* afp, const
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
float* ferr, float* berr );
```

```c
lapack_int LAPACKE_zpprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
const lapack_complex_double* ap, const lapack_complex_double* afp, const
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations \( A \times X = B \) with a symmetric (Hermitian) positive definite matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( x \) is the exact solution of the perturbed system:

\[
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A) x = (b + \delta b).
\]

Finally, the routine estimates the component-wise forward error in the computed solution

\[
\|x - x_e\|_\infty / \|x\|_\infty
\]

where \( x_e \) is the exact solution.

Before calling this routine:

- call the factorization routine ?pptrf
- call the solver routine ?pptrs.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.

  Indicates how the input matrix \( A \) has been factored:
  - If \( \text{uplo} = 'U' \), the upper triangle of \( A \) is stored.
  - If \( \text{uplo} = 'L' \), the lower triangle of \( A \) is stored.
The order of the matrix \( A \); \( n \geq 0 \).

The number of right-hand sides; \( nrhs \geq 0 \).

\( ap \) contains the original matrix \( A \) in a packed format, as supplied to \(?pptrf\). The dimension of \( ap \) must be at least \( \max(1, n(n+1)/2) \).

\( afp \) contains the factored matrix \( A \) in a packed format, as returned by \(?pptrf\). The dimension of \( afp \) must be at least \( \max(1, n(n+1)/2) \).

Array \( b \) of size \( \max(1, ldb*nrhs) \) for column major layout and \( \max(1, ldb*n) \) for row major layout contains the right-hand side matrix \( B \).

Array \( x \) of size \( \max(1, ldx*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout contains the solution matrix \( X \).

The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

The leading dimension of \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

**Output Parameters**

\( x \) The refined solution matrix \( X \).

\( ferr, berr \) Arrays, size at least \( \max(1, nrhs) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

**Application Notes**

The bounds returned in \( ferr \) are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of \( 4n^2 \) floating-point operations (for real flavors) or \( 16n^2 \) operations (for complex flavors). In addition, each step of iterative refinement involves \( 6n^2 \) operations (for real flavors) or \( 24n^2 \) operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations \( A^*x = b \); the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately \( 2n^2 \) floating-point operations for real flavors or \( 8n^2 \) for complex flavors.

**See Also**

Matrix Storage Schemes

\(?pbrfs\)

Refines the solution of a system of linear equations with a band symmetric (Hermitian) positive-definite coefficient matrix and estimates its error.
Syntax

```
lapack_int LAPACKE_spbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd, 
                      lapack_int nrhs, const float* ab, lapack_int ldab, const float* afb, lapack_int ldafb, 
                      const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_dpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd, 
                      lapack_int nrhs, const double* ab, lapack_int ldab, const double* afb, lapack_int ldafb, 
                      const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```
lapack_int LAPACKE_cpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd, 
                      lapack_int nrhs, const lapack_complex_float* ab, lapack_int ldab, const 
                      lapack_complex_float* afb, lapack_int ldafb, const lapack_complex_float* b, lapack_int 
                      ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```
lapack_int LAPACKE_zpbrfs( int matrix_layout, char uplo, lapack_int n, lapack_int kd, 
                      lapack_int nrhs, const lapack_complex_double* ab, lapack_int ldab, const 
                      lapack_complex_double* afb, lapack_int ldafb, const lapack_complex_double* b, lapack_int 
                      ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations \( A X = B \) with a symmetric (Hermitian) positive definite band matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( x \) is the exact solution of the perturbed system:

\[
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).
\]

Finally, the routine estimates the component-wise forward error in the computed solution \( ||x - x_e||_\infty / ||x||_\infty \) (here \( x_e \) is the exact solution).

Before calling this routine:
- call the factorization routine \(?pbtrf\)
- call the solver routine \(?pbtrs\).

Input Parameters

- `matrix_layout`: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo`: Must be 'U' or 'L'. Indicates how the input matrix \( A \) has been factored:
  - If `uplo` = 'U', the upper triangle of \( A \) is stored.
  - If `uplo` = 'L', the lower triangle of \( A \) is stored.
- `n`: The order of the matrix \( A \); \( n \geq 0 \).
- `kd`: The number of superdiagonals or subdiagonals in the matrix \( A \); \( kd \geq 0 \).
- `nrhs`: The number of right-hand sides; \( nrhs \geq 0 \).
$ab$

Array $ab$ (size max($ldab\times n$)) contains the original band matrix $A$, as supplied to ?pbtrf.

$afb$

Array $afb$ (size max($ldafb\times n$)) contains the factored band matrix $A$, as returned by ?pbtrf.

$b$

Array $b$ of size max($1, ldb\times nrhs$) for column major layout and max($1, ldb\times n$) for row major layout contains the right-hand side matrix $B$.

$x$

Array $x$ of size max($1, ldx\times nrhs$) for column major layout and max($1, ldx\times n$) for row major layout contains the solution matrix $X$.

$ldab$

The leading dimension of $ab$; $ldab\geq kd + 1$.

$ldafb$

The leading dimension of $afb$; $ldafb\geq kd + 1$.

$ldb$

The leading dimension of $b$; $ldb\geq\text{max}(1, n)$ for column major layout and $ldb\geq\text{nrhs}$ for row major layout.

$ldx$

The leading dimension of $x$; $ldx\geq\text{max}(1, n)$ for column major layout and $ldx\geq\text{nrhs}$ for row major layout.

Output Parameters

$x$

The refined solution matrix $X$.

$ferr, berr$

Arrays, size at least max($1, nrhs$). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value $info$. If $info = 0$, the execution is successful. If $info = -i$, parameter $i$ had an illegal value.

Application Notes

The bounds returned in $ferr$ are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $8n\times kd$ floating-point operations (for real flavors) or $32n\times kd$ operations (for complex flavors). In addition, each step of iterative refinement involves $12n\times kd$ operations (for real flavors) or $48n\times kd$ operations (for complex flavors); the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $4n\times kd$ floating-point operations for real flavors or $16n\times kd$ for complex flavors.

See Also

Matrix Storage Schemes

?ptrfs

Refines the solution of a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal coefficient matrix and estimates its error.
Syntax

```c
lapack_int LAPACKE_sptrfs( int matrix_layout, lapack_int n, lapack_int nrhs, const float* d, const float* e, const float* df, const float* ef, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dptrfs( int matrix_layout, lapack_int n, lapack_int nrhs, const double* d, const double* e, const double* df, const double* ef, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_cptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float* d, const lapack_complex_float* e, const float* df, const lapack_complex_float* ef, const float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_zptrfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double* d, const lapack_complex_double* e, const double* df, const lapack_complex_double* ef, const double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A \cdot X = B$ with a symmetric (Hermitian) positive definite tridiagonal matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error $\beta$. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i|$$

such that $(A + \delta A) \cdot x = (b + \delta b)$.

Finally, the routine estimates the component-wise forward error in the computed solution $\|x - x_e\|/\|x\|$ (here $x_e$ is the exact solution).

Before calling this routine:

- call the factorization routine $\texttt{?pttrf}$
- call the solver routine $\texttt{?pttrs}$.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Used for complex flavors only. Must be 'U' or 'L'.
  Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix $A$ is stored and how $A$ is factored:
  - If $\text{uplo} = 'U'$, the array $e$ stores the superdiagonal of $A$, and $A$ is factored as $U^H \cdot D \cdot U$.
  - If $\text{uplo} = 'L'$, the array $e$ stores the subdiagonal of $A$, and $A$ is factored as $L \cdot D \cdot L^H$.

- **n**
  The order of the matrix $A; n \geq 0$.

- **nrhs**
  The number of right-hand sides; $nrhs \geq 0$. 

553
The array \( d \) (size \( n \)) contains the \( n \) diagonal elements of the tridiagonal matrix \( A \).

The array \( df \) (size \( n \)) contains the \( n \) diagonal elements of the diagonal matrix \( D \) from the factorization of \( A \) as computed by \(?pttrf\).

The array \( e \) (size \( n - 1 \)) contains the \((n - 1)\) off-diagonal elements of the tridiagonal matrix \( A \) (see \( uplo \)).

The array \( ef \) (size \( n - 1 \)) contains the \((n - 1)\) off-diagonal elements of the unit bidiagonal factor \( U \) or \( L \) from the factorization computed by \(?pttrf\) (see \( uplo \)).

The array \( b \) of size \( \max(1, ldb*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

The array \( x \) of size \( \max(1, ldx*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout contains the solution matrix \( X \) as computed by \(?pttrs\).

The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

The leading dimension of \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

Output Parameters

- \( x \): The refined solution matrix \( X \).
- \( ferr, berr \): Arrays, size at least \( \max(1, nrhs) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

See Also

Matrix Storage Schemes

\(?syrs\)

Refines the solution of a system of linear equations with a symmetric coefficient matrix and estimates its error.

Syntax

```c
lapack_int LAPACKE_ssyrs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```
The routine performs an iterative refinement of the solution to a system of linear equations \( A^*X = B \) with a symmetric full-storage matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( x \) is the exact solution of the perturbed system:

\[
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).
\]

Finally, the routine estimates the component-wise forward error in the computed solution \( || x - x_e ||_\infty / || x ||_\infty \) (here \( x_e \) is the exact solution).

Before calling this routine:
- call the factorization routine \(?sytrf\)
- call the solver routine \(?sytrs\).

### Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.

  - If \( uplo = 'U' \), the upper triangle of \( A \) is stored.
  - If \( uplo = 'L' \), the lower triangle of \( A \) is stored.

- **n**
  The order of the matrix \( A \); \( n \geq 0 \).

- **nrhs**
  The number of right-hand sides; \( nrhs \geq 0 \).

- **a**
  Array of size \( \text{max}(1, lda*n) \) contains the original matrix \( A \), as supplied to \(?sytrf\).

- **af**
  Array \( af \) of size \( \text{max}(1, ldaf*n) \) contains the factored matrix \( A \), as returned by \(?sytrf\).

- **b**
  Array \( b \) of size \( \text{max}(1, ldb*nrhs) \) for column major layout and \( \text{max}(1, ldb*n) \) for row major layout contains the right-hand side matrix \( B \).
Array $x$ of size $\max(1, ldx\times nrhs)$ for column major layout and $\max(1, ldx\times n)$ for row major layout contains the solution matrix $X$.

The leading dimension of $a$; $lda \geq \max(1, n)$.

The leading dimension of $af$; $ldaf \geq \max(1, n)$.

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

The leading dimension of $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Array, size at least $\max(1, n)$. The $ipiv$ array, as returned by $\text{?sytrf}$.

Output Parameters

$x$ The refined solution matrix $X$.

$ferr, berr$ Arrays, size at least $\max(1, nrhs)$. Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

Application Notes

The bounds returned in $ferr$ are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5. Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also

Matrix Storage Schemes

$\text{?syrsx}$

Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix $A$ and provides error bounds and backward error estimates.

Syntax

```c
lapack_int LAPACKE_ssyrfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );
```
lapack_int LAPACKE_dsyrfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

lapack_int LAPACKE_csyrfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zsyrfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );

Include Files

• mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is symmetric indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for err_bnds_norm and err_bnds_comp for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters equed and s below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

matrix_layout

Specifies whether two-dimensional array storage is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored:

If uplo = 'U', the upper triangle of A is stored.

If uplo = 'L', the lower triangle of A is stored.

equed

Must be 'N' or 'Y'.

Specifies the form of equilibration that was done to A before calling this routine.

If equed = 'N', no equilibration was done.

If equed = 'Y', both row and column equilibration was done, that is, A has been replaced by diag(s)*A*diag(s). The right-hand side B has been changed accordingly.

n

The number of linear equations; the order of the matrix A; \( n \geq 0 \).
The number of right-hand sides; the number of columns of the matrices \( B \) and \( X \); \( nrhs \geq 0 \).

The array \( a \) (size \( \max(1, lda*n) \)) contains the symmetric/Hermitian matrix \( A \) as specified by \( uplo \). If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( a \) contains the upper triangular part of the matrix \( A \) and the strictly lower triangular part of \( a \) is not referenced. If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( a \) contains the lower triangular part of the matrix \( A \) and the strictly upper triangular part of \( a \) is not referenced.

The array \( af \) (size \( \max(1, ldaf*n) \)) contains the triangular factor \( L \) or \( U \) from the Cholesky factorization \( A = U^T U \) or \( A = L L^T \) as computed by \texttt{ssytrf} for real flavors or \texttt{dsytrf} for complex flavors.

The array \( b \) (size \( \max(1, ldb*nrhs) \) for column major layout and \( \max(1, ldb*n) \) for row major layout) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

The leading dimension of \( a \); \( lda \geq \max(1, n) \).

The leading dimension of \( af \); \( ldaf \geq \max(1, n) \).

Array, size at least \( \max(1, n) \). Contains details of the interchanges and the block structure of \( D \) as determined by \texttt{ssytrf} for real flavors or \texttt{dsytrf} for complex flavors.

Array, size \( (n) \). The array \( s \) contains the scale factors for \( A \).

If \( equed = 'N' \), \( s \) is not accessed.

If \( equed = 'Y' \), each element of \( s \) must be positive.

Each element of \( s \) should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Array, of size \( \max(1, ldx*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout.

The solution matrix \( X \) as computed by \texttt{?sytrs}

The leading dimension of the output array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See \texttt{err_bnds_norm} and \texttt{err_bnds_comp} descriptions in \texttt{Output Arguments} section below.

Specifies the number of parameters set in \( \texttt{params} \). If \( \leq 0 \), the \( \texttt{params} \) array is never referenced and default values are used.
Array, size \texttt{np}arams. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to \texttt{np}arams are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass \texttt{np}arams = 0, which prevents the source code from accessing the \texttt{params} argument.

\texttt{params}[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0
No refinement is performed and no error bounds are computed.

=1.0
Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

\texttt{params}[1] : Maximum number of residual computations allowed for refinement.

Default 10.0
Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than \textit{LU}. If the factorization uses a technique other than Gaussian elimination, the guarantees in \texttt{err_bnds_norm} and \texttt{err_bnds_comp} may no longer be trustworthy.

\texttt{params}[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

\textbf{Output Parameters}

\textit{x}
The improved solution matrix \( X \).

\textit{rcond}
Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix \( A \) after equilibration (if done). If \texttt{rcond} is less than the machine precision, in particular, if \texttt{rcond} = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

\textit{berr}
Array, size at least \( \max(1, \texttt{nrhs}) \). Contains the componentwise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of \( A \) or \( B \) that makes  \( x_j \) an exact solution.

\textit{err_bnds_norm}
Array of size \( \texttt{nrhs} \times \texttt{n_{err_bnds}} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the \( i \)-th solution vector
The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

**err=1**  
"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors.

**err=2**  
"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

**err=3**  
Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error $\text{err}$ is stored in:

- Column major layout: $\text{err}_\text{bnds}_\text{norm}[(\text{err} - 1) \times \text{nrhs} + i - 1]$.
- Row major layout: $\text{err}_\text{bnds}_\text{norm}[	ext{err} - 1 + (i - 1) \times n_{\text{err}_\text{bnds}}]$.

Array of size $\text{nrhs} \times n_{\text{err}_\text{bnds}}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the $i$-th solution vector:

$$\max_j \frac{|X_{\text{true}_{ji}} - X_{\text{ji}}|}{\max_j |X_{\text{ji}}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params}[2] = 0.0$), then $\text{err}_\text{bnds}_\text{comp}$ is not accessed.
err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \( Z \) are:

\[
\frac{\|Z\|_\infty}{\|Z^{-1}\|_\infty}
\]

Let \( z = s \cdot (a \cdot \text{diag}(x)) \), where \( x \) is the solution for the current right-hand side and \( s \) scales each row of \( a \cdot \text{diag}(x) \) by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in:

- Column major layout: \( \text{err_bnds_comp}[(\text{err} - 1) \times \text{nrhs} + i - 1] \).
- Row major layout: \( \text{err_bnds_comp}[(\text{err} - 1) + (i - 1) \times \text{n_err_bnds}] \)

\( \text{params} \)

Output parameter only if the input contains erroneous values, namely, in \( \text{params}[0], \text{params}[1], \text{params}[2] \). In such a case, the corresponding elements of \( \text{params} \) are filled with default values on output.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( 0 < \text{info} < n \): \( U_{i\text{\text{info}},i\text{\text{info}}} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( rcond = 0 \) is returned.

If \( \text{info} = n+j \): The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th
right-hand side is the first with a normwise error bound that is not guaranteed (the smallest $j$ such that for column major layout $err_bnds_norm[j-1] = 0.0$ or $err_bnds_comp[j-1] = 0.0$; or for row major layout $err_bnds_norm[(j-1)*n + err_bnds] = 0.0$ or $err_bnds_comp[(j-1)*n + err_bnds] = 0.0$). See the definition of $err_bnds_norm$ and $err_bnds_comp$ for $err = 1$. To get information about all of the right-hand sides, check $err_bnds_norm$ or $err_bnds_comp$.

See Also
Matrix Storage Schemes

?herfs
Refines the solution of a system of linear equations with a complex Hermitian coefficient matrix and estimates its error.

Syntax

```c
lapack_int LAPACKE_cherfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
 const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af,
 lapack_int ldaf, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
 lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

Include Files
- mkl.h

Description
The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a complex Hermitian full-storage matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error $\beta$. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).
$$

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty/||x||_\infty$ (here $x_e$ is the exact solution).

Before calling this routine:
- call the factorization routine ?hetrf
- call the solver routine ?hetrs.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  If uplo = 'U', the upper triangle of $A$ is stored.
  If uplo = 'L', the lower triangle of $A$ is stored.

- **n**
  The order of the matrix $A$; $n \geq 0$. 
nrhs

The number of right-hand sides; \( nrhs \geq 0 \).

Arrays:

\( a \) (size max(1, \( lda * n \))) contains the original matrix \( A \), as supplied to \( ?hetrf \).

\( af \) (size max(1, \( ldaf * n \))) contains the factored matrix \( A \), as returned by \( ?hetrf \).

\( b \) of size max(1, \( ldb * nrhs \)) for column major layout and max(1, \( ldb * n \)) for row major layout contains the right-hand side matrix \( B \).

\( x \) of size max(1, \( ldx * nrhs \)) for column major layout and max(1, \( ldx * n \)) for row major layout contains the solution matrix \( X \).

\( lda \)

The leading dimension of \( a \); \( lda \geq \) max(1, \( n \)).

\( ldaf \)

The leading dimension of \( af \); \( ldaf \geq \) max(1, \( n \)).

\( ldb \)

The leading dimension of \( b \); \( ldb \geq \) max(1, \( n \)) for column major layout and \( \geq \) \( nrhs \) for row major layout.

\( ldx \)

The leading dimension of \( x \); \( ldx \geq \) max(1, \( n \)) for column major layout and \( \geq \) \( nrhs \) for row major layout.

\( ipiv \)

Array, size at least \( \max(1, n) \). The \( ipiv \) array, as returned by \( ?hetrf \).

Output Parameters

\( x \)

The refined solution matrix \( X \).

\( ferr, berr \)

Arrays, size at least \( \max(1, nrhs) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes

The bounds returned in \( ferr \) are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of \( 16n^2 \) operations. In addition, each step of iterative refinement involves \( 24n^2 \) operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations \( A^*x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 8n^2 \) floating-point operations.

The real counterpart of this routine is \( ?ssyrsf/\?dsyrsf \)

See Also

Matrix Storage Schemes
Uses extra precise iterative refinement to improve the solution to the system of linear equations with a symmetric indefinite coefficient matrix A and provides error bounds and backward error estimates.

Syntax

```c
lapack_int LAPACKE_cherfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* af, lapack_int ldaf, const lapack_int* ipiv, const float* s, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, float* params );

lapack_int LAPACKE_zherfsx( int matrix_layout, char uplo, char equed, lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* af, lapack_int ldaf, const lapack_int* ipiv, const double* s, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, double* params );
```

Include Files

- mkl.h

Description

The routine improves the computed solution to a system of linear equations when the coefficient matrix is Hermitian indefinite, and provides error bounds and backward error estimates for the solution. In addition to a normwise error bound, the code provides a maximum componentwise error bound, if possible. See comments for `err_bnds_norm` and `err_bnds_comp` for details of the error bounds.

The original system of linear equations may have been equilibrated before calling this routine, as described by the parameters `equed` and `s` below. In this case, the solution and error bounds returned are for the original unequilibrated system.

Input Parameters

- **matrix_layout**
  Specifies whether two-dimensional array storage is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **uplo**
  Must be `'U'` or `'L'`.
  Indicates whether the upper or lower triangular part of A is stored:
  - If `uplo` = `'U'`, the upper triangle of A is stored.
  - If `uplo` = `'L'`, the lower triangle of A is stored.

- **equed**
  Must be `'N'` or `'Y'`.
  Specifies the form of equilibration that was done to A before calling this routine:
  - If `equed` = `'N'`, no equilibration was done.
  - If `equed` = `'Y'`, both row and column equilibration was done, that is, A has been replaced by `diag(s)*A*diag(s)`. The right-hand side B has been changed accordingly.
The number of linear equations; the order of the matrix $A$; \( n \geq 0 \).

The number of right-hand sides; the number of columns of the matrices $B$ and $X$; \( nrhs \geq 0 \).

The array $a$ of size $\max(1, \text{lda} \times n)$ contains the Hermitian matrix $A$ as specified by $\text{uplo}$. If $\text{uplo} = 'U'$, the leading $n$-by-$n$ upper triangular part of $a$ contains the upper triangular part of the matrix $A$ and the strictly lower triangular part of $a$ is not referenced. If $\text{uplo} = 'L'$, the leading $n$-by-$n$ lower triangular part of $a$ contains the lower triangular part of the matrix $A$ and the strictly upper triangular part of $a$ is not referenced.

The array $af$ of size $\max(1, \text{ldaf} \times n)$ contains the block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ or $L$ from the factorization $A = U^*D*U^T$ or $A = L^*D*L^T$ as computed by $\text{ssytrf}$ for $\text{cherfsx}$ or $\text{dsytrf}$ for $\text{zherfsx}$.

The array $b$ of size $\max(1, \text{ldb} \times nrhs)$ for row major layout and $\max(1, \text{ldb} \times n)$ for column major layout contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

The leading dimension of $a$; \( \text{lda} \geq \max(1, n) \).

The leading dimension of $af$; \( \text{ldaf} \geq \max(1, n) \).

Array, size at least $\max(1, n)$. Contains details of the interchanges and the block structure of $D$ as determined by $\text{ssytrf}$ for real flavors or $\text{dsytrf}$ for complex flavors.

Array, size $(n)$. The array $s$ contains the scale factors for $A$.

If $\text{equed} = 'N'$, $s$ is not accessed.

If $\text{equed} = 'Y'$, each element of $s$ must be positive.

Each element of $s$ should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

The leading dimension of the array $b$; \( \text{ldb} \geq \max(1, n) \) for column major layout and $\text{ldb} \geq nrhs$ for row major layout.

Array, size $\max(1, \text{ldx} \times nrhs)$ for column major layout and $\max(1, \text{ldx} \times n)$ for row major layout.

The solution matrix $X$ as computed by $\text{?hetrs}$.

The leading dimension of the output array $x$; \( \text{ldx} \geq \max(1, n) \) for column major layout and $\text{ldx} \geq nrhs$ for row major layout.

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See $\text{err_bnds_norm}$ and $\text{err_bnds_comp}$ descriptions in $\text{Output Arguments}$ section below.

Specifies the number of parameters set in $\text{params}$. If \( \leq 0 \), the $\text{params}$ array is never referenced and default values are used.
**params**

Array, size nparams. Specifies algorithm parameters. If an entry is less than 0.0, that entry will be filled with the default value used for that parameter. Only positions up to nparams are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass nparams = 0, which prevents the source code from accessing the params argument.

**params[0]**: Whether to perform iterative refinement or not. Default: 1.0 (for cherfsx), 1.0D+0 (for zherfsx).

- **0.0**: No refinement is performed and no error bounds are computed.
- **1.0**: Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

**params[1]**: Maximum number of residual computations allowed for refinement.

- **Default**: 10
- **Aggressive**: Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in err_bnds_norm and err_bnds_comp may no longer be trustworthy.

**params[2]**: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

**Output Parameters**

- **x**: The improved solution matrix X.
- **rcond**: Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix A after equilibration (if done). If rcond is less than the machine precision, in particular, if rcond = 0, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

- **berr**: Array, size at least max(1, nrhs). Contains the componentwise relative backward error for each solution vector xj, that is, the smallest relative change in any element of A or B that makes xj an exact solution.

- **err_bnds_norm**: Array of size nrhs*n_err_bnds. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

  Normwise relative error in the i-th solution vector
The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

\[ \text{err} = 1 \]

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for \texttt{cherfsx} and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for \texttt{zherfsx}.

\[ \text{err} = 2 \]

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for \texttt{cherfsx} and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for \texttt{zherfsx}. This error bound should only be trusted if the previous boolean is true.

\[ \text{err} = 3 \]

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \( Z \) are:

\[ \|Z\|_\infty \cdot \|Z^{-1}\|_\infty \]

Let \( z = s \cdot a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in:

- **Column major layout**: \( \text{err}_{\text{bnds}}_{\text{norm}}[(\text{err} - 1) \times \text{nrhs} + i - 1] \).
- **Row major layout**: \( \text{err}_{\text{bnds}}_{\text{norm}}[\text{err} - 1 + (i - 1) \times \text{n_err_bnds}] \)

\( \text{err}_{\text{bnds}}_{\text{comp}} \)

Array of size \( \text{nrhs} \times \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \( i \)-th solution vector:

\[ \max_j \frac{|X_{\text{true}_{ji}} - X_{ji}|}{|X_{ji}|} \]
The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (\texttt{params[2]} = 0.0), then \texttt{err_bnds_comp} is not accessed.

\texttt{err=1} "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \(\sqrt{n} \times \text{slamch}(\varepsilon)\) for \texttt{cherfsx} and \(\sqrt{n} \times \text{dlamch}(\varepsilon)\) for \texttt{zherfsx}.

\texttt{err=2} "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \(\sqrt{n} \times \text{slamch}(\varepsilon)\) for \texttt{cherfsx} and \(\sqrt{n} \times \text{dlamch}(\varepsilon)\) for \texttt{zherfsx}. This error bound should only be trusted if the previous boolean is true.

\texttt{err=3} Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold \(\sqrt{n} \times \text{slamch}(\varepsilon)\) for single precision flavors and \(\sqrt{n} \times \text{dlamch}(\varepsilon)\) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \(Z\) are:

\[
\begin{align*}
\|Z\|_\infty & \cdot \|Z^{-1}\|_\infty
\end{align*}
\]

Let \(z=s \times (a \times \text{diag}(x))\), where \(x\) is the solution for the current right-hand side and \(s\) scales each row of \(a \times \text{diag}(x)\) by a power of the radix so all absolute row sums of \(z\) are approximately 1.

The information for right-hand side \(i\), where \(1 \leq i \leq \texttt{nrhs}\), and type of error \texttt{err} is stored in:

- Column major layout: \texttt{err_bnds_comp[(err - 1) \times \texttt{nrhs} + i - 1]}.
- Row major layout: \texttt{err_bnds_comp[err - 1 + (i - 1) \times \texttt{n_err_bnds}]}.

\texttt{params}

Output parameter only if the input contains erroneous values, namely, in \texttt{params[0]}, \texttt{params[1]}, \texttt{params[2]}. In such a case, the corresponding elements of \texttt{params} are filled with default values on output.

Return Values

This function returns a value \texttt{info}.

If \(\texttt{info} = 0\), the execution is successful. The solution to every right-hand side is guaranteed.

If \(\texttt{info} = -i\), parameter \(i\) had an illegal value.

If \(0 < \texttt{info}\): \(\texttt{info}\) is exactly zero. The factorization has been completed, but the factor \(D\) is exactly singular, so the solution and error bounds could not be computed; \(\texttt{rcond} = 0\) is returned.
If $info = n+j$: The solution corresponding to the $j$-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides $k$ with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $params[2] = 0.0$, then the $j$-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest $j$ such that for column major layout $err\_bnds\_norm[j-1] = 0.0$ or $err\_bnds\_comp[j-1] = 0.0$; or for row major layout $err\_bnds\_norm[(j-1)*n\_err\_bnds] = 0.0$ or $err\_bnds\_comp[(j-1)*n\_err\_bnds] = 0.0$). See the definition of $err\_bnds\_norm$ and $err\_bnds\_comp$ for $err = 1$. To get information about all of the right-hand sides, check $err\_bnds\_norm$ or $err\_bnds\_comp$.

See Also
Matrix Storage Schemes

?sprfs
Refines the solution of a system of linear equations with a packed symmetric coefficient matrix and estimates the solution error.

Syntax

```c
lapack_int LAPACKE_ssprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const float* ap, const float* afp, const lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* ferr, float* berr );
```

```c
lapack_int LAPACKE_dsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const double* ap, const double* afp, const lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* ferr, double* berr );
```

```c
lapack_int LAPACKE_csprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr );
```

```c
lapack_int LAPACKE_zsprfs( int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr );
```

Include Files
- mkl.h

Description
The routine performs an iterative refinement of the solution to a system of linear equations $A*X = B$ with a packed symmetric matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error $β$. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).$$

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here $x_e$ is the exact solution).

Before calling this routine:
- call the factorization routine ?sptrfs
- call the solver routine ?sprtrs.
**Input Parameters**

*matrix_layout*  
Specifies whether matrix storage layout is row major (*LAPACK_ROW_MAJOR*) or column major (*LAPACK_COL_MAJOR*).

*uplo*  
Must be 'U' or 'L'.  
If *uplo* = 'U', the upper triangle of *A* is stored.  
If *uplo* = 'L', the lower triangle of *A* is stored.

*n*  
The order of the matrix *A*; *n* ≥ 0.

*nrhs*  
The number of right-hand sides; *nrhs* ≥ 0.

*ap, afp, b, x*  
Arrays:  
*ap* of size max(1, *n*(n+1)/2) contains the original packed matrix *A*, as supplied to ?sptrf.  
*afp* of size max(1, *n*(n+1)/2) contains the factored packed matrix *A*, as returned by ?sptrf.  
*b* of size max(1, ldb*nrhs) for column major layout and max(1, ldx*n) for row major layout contains the right-hand side matrix *B*.  
*x* of size max(1, ldx*nrhs) for column major layout and max(1, ldx*n) for row major layout contains the solution matrix *X*.

*ldb*  
The leading dimension of *b*; *ldb* ≥ max(1, *n*) for column major layout and *ldb* ≥ *nrhs* for row major layout.

*ldx*  
The leading dimension of *x*; *ldx* ≥ max(1, *n*) for column major layout and *ldx* ≥ max(1, *nrhs*) for row major layout.

*ipiv*  
Array, size at least max(1, *n*). The *ipiv* array, as returned by ?sptrf.

**Output Parameters**

*x*  
The refined solution matrix *X*.

*ferr, berr*  
Arrays, size at least max(1, *nrhs*). Contain the component-wise forward and backward errors, respectively, for each solution vector.

**Return Values**

This function returns a value *info*.  
If *info* = 0, the execution is successful.  
If *info* = -i, parameter *i* had an illegal value.

**Application Notes**

The bounds returned in *ferr* are not rigorous, but in practice they almost always overestimate the actual error.  
For each right-hand side, computation of the backward error involves a minimum of $4n^2$ floating-point operations (for real flavors) or $16n^2$ operations (for complex flavors). In addition, each step of iterative refinement involves $6n^2$ operations (for real flavors) or $24n^2$ operations (for complex flavors); the number of iterations may range from 1 to 5.
Estimating the forward error involves solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n^2$ floating-point operations for real flavors or $8n^2$ for complex flavors.

See Also
Matrix Storage Schemes

?hprfs
Refines the solution of a system of linear equations with a packed complex Hermitian coefficient matrix and estimates the solution error.

Syntax

```c
lapack_int LAPACKE_chiprfs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float* ap, const lapack_complex_float* afp, const lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr);

lapack_int LAPACKE_zhiprfs(int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double* ap, const lapack_complex_double* afp, const lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr);
```

Include Files

- mkl.h

Description

The routine performs an iterative refinement of the solution to a system of linear equations $A^*X = B$ with a packed complex Hermitian matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A)x = (b + \delta b).
$$

Finally, the routine estimates the component-wise forward error in the computed solution $||x - x_e||_{\infty}/||x||_{\infty}$ (here $x_e$ is the exact solution).

Before calling this routine:
- call the factorization routine ?hptrf
- call the solver routine ?hptrs.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'.
  - If uplo = 'U', the upper triangle of $A$ is stored.
  - If uplo = 'L', the lower triangle of $A$ is stored.
- **n**: The order of the matrix $A$; $n \geq 0$.
- **nrhs**: The number of right-hand sides; $nrhs \geq 0$. 
Arrays:
ap\text{max}(1, n(n + 1)/2) contains the original packed matrix \( A \), as supplied to \( \text{?hptrf} \).
afp\text{max}(1, n(n + 1)/2) contains the factored packed matrix \( A \), as returned by \( \text{?hptrf} \).

b of size \text{max}(1, ldb*nrhs) for column major layout and \text{max}(1, ldx*n) for row major layout contains the right-hand side matrix \( B \).
x of size \text{max}(1, ldx*nrhs) for column major layout and \text{max}(1, ldb*n) for row major layout contains the solution matrix \( X \).

\text{ldb}
The leading dimension of \( b \); \( ldb \geq \text{max}(1, n) \) for column major layout and \( ldb \geq \text{nrhs} \) for row major layout.

\text{ldx}
The leading dimension of \( x \); \( ldx \geq \text{max}(1, n) \) for column major layout and \( ldx \geq \text{nrhs} \) for row major layout.

ipiv
Array, size at least \text{max}(1, n). The \( ipiv \) array, as returned by \( \text{?hptrf} \).

Output Parameters
x
The refined solution matrix \( X \).

ferr, berr
Arrays, size at least \text{max}(1, nrhs). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values
This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), parameter \( i \) had an illegal value.

Application Notes
The bounds returned in \( ferr \) are not rigorous, but in practice they almost always overestimate the actual error.

For each right-hand side, computation of the backward error involves a minimum of \( 16n^2 \) operations. In addition, each step of iterative refinement involves \( 24n^2 \) operations; the number of iterations may range from 1 to 5.

Estimating the forward error involves solving a number of systems of linear equations \( A^*x = b \); the number is usually 4 or 5 and never more than 11. Each solution requires approximately \( 8n^2 \) floating-point operations.

The real counterpart of this routine is \( \text{?ssprfs} / \text{?dsprfs} \).

See Also
Matrix Storage Schemes

\( \text{?trrfs} \)
Estimates the error in the solution of a system of linear equations with a triangular coefficient matrix.
**Syntax**

```c
lapack_int LAPACKE_strrfs( int matrix_layout, char uplo, char trans, char diag,
                      lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, const float* b,
                      lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dtrrfs( int matrix_layout, char uplo, char trans, char diag,
                      lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, const double* b,
                      lapack_int ldb, const double* x, lapack_int ldx, double* ferr, double* berr );

lapack_int LAPACKE_ctrrfs( int matrix_layout, char uplo, char trans, char diag,
                      lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, const
                      lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* x, lapack_int ldx,
                      float* ferr, float* berr );

lapack_int LAPACKE_ztrrfs( int matrix_layout, char uplo, char trans, char diag,
                      lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, const
                      lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* x, lapack_int ldx,
                      double* ferr, double* berr );
```

**Include Files**

- mkl.h

**Description**

The routine estimates the errors in the solution to a system of linear equations \( A \times X = B \) or \( A^T \times X = B \) or \( A^H \times X = B \) with a triangular matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the *component-wise backward error*. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \( x \) is the exact solution of the perturbed system:

\[
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A) x = (b + \delta b).
\]

The routine also estimates the *component-wise forward error* in the computed solution \( ||x - x_e||_\infty/||x||_\infty \) (here \( x_e \) is the exact solution).

Before calling this routine, call the solver routine ?trtrs.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'. Indicates whether \( A \) is upper or lower triangular:
  - If `uplo` = 'U', then \( A \) is upper triangular.
  - If `uplo` = 'L', then \( A \) is lower triangular.

- **trans**
  Must be 'N' or 'T' or 'C'. Indicates the form of the equations:
  - If `trans` = 'N', the system has the form \( A \times X = B \).
  - If `trans` = 'T', the system has the form \( A^T \times X = B \).
  - If `trans` = 'C', the system has the form \( A^H \times X = B \).

- **diag**
  Must be 'N' or 'U'.
If \( \text{diag} = 'N' \), then \( A \) is not a unit triangular matrix.

If \( \text{diag} = 'U' \), then \( A \) is unit triangular: diagonal elements of \( A \) are assumed to be 1 and not referenced in the array \( a \).

\( n \)

The order of the matrix \( A \); \( n \geq 0 \).

\( \text{nrhs} \)

The number of right-hand sides; \( \text{nrhs} \geq 0 \).

\( a, b, x \)

Arrays:

\( a(\text{size } \max(1, \text{lda} \times n)) \) contains the upper or lower triangular matrix \( A \), as specified by \( \text{uplo} \).

\( b \) of size \( \max(1, \text{ldb} \times \text{nrhs}) \) for column major layout and \( \max(1, \text{ldb} \times n) \) for row major layout contains the right-hand side matrix \( B \).

\( x \) of size \( \max(1, \text{ldx} \times \text{nrhs}) \) for column major layout and \( \max(1, \text{ldx} \times n) \) for row major layout contains the solution matrix \( X \).

\( \text{lda} \)

The leading dimension of \( a \); \( \text{lda} \geq \max(1, n) \).

\( \text{ldb} \)

The leading dimension of \( b \); \( \text{ldb} \geq \max(1, n) \) for column major layout and \( \text{ldb} \geq \text{nrhs} \) for row major layout.

\( \text{ldx} \)

The leading dimension of \( x \); \( \text{ldx} \geq \max(1, n) \) for column major layout and \( \text{ldx} \geq \text{nrhs} \) for row major layout.

**Output Parameters**

\( \text{ferr}, \text{berr} \)

Arrays, size at least \( \max(1, \text{nrhs}) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

**Application Notes**

The bounds returned in \( \text{ferr} \) are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations \( A^*x = b \); the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately \( n^2 \) floating-point operations for real flavors or \( 4n^2 \) for complex flavors.

**See Also**

**Matrix Storage Schemes**

?tprfs

*Estimates the error in the solution of a system of linear equations with a packed triangular coefficient matrix.*
Syntax

```c
lapack_int LAPACKE_stprfs( int matrix_layout, char uplo, char trans, char diag,
                           int n, int nrhs, const float* ap, const float* b, int ldb,
                           const float* x, int ldx, float* ferr, float* berr );

lapack_int LAPACKE_dtptrfs( int matrix_layout, char uplo, char trans, char diag,
                            int n, int nrhs, const double* ap, const double* b, int ldb,
                            const double* x, int ldx, double* ferr, double* berr );

lapack_int LAPACKE_ctprfs( int matrix_layout, char uplo, char trans, char diag,
                           int n, int nrhs, const lapack_complex_float* ap,
                           const lapack_complex_float* b, int ldb,
                           const lapack_complex_float* x, int ldx, float* ferr,
                           float* berr );

lapack_int LAPACKE_ztprfs( int matrix_layout, char uplo, char trans, char diag,
                           int n, int nrhs, const lapack_complex_double* ap,
                           const lapack_complex_double* b, int ldb,
                           const lapack_complex_double* x, int ldx,
                           double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations \( AX = B \) or \( A^T X = B \) or \( A^H X = B \) with a packed triangular matrix \( A \), with multiple right-hand sides. For each computed solution vector \( x \), the routine computes the **component-wise backward error**. This error is the smallest relative perturbation in elements of \( A \) and \( b \) such that \((A + \delta A)x = (b + \delta b)\).

The routine also estimates the **component-wise forward error** in the computed solution \( \| x - x_e \|_\infty / \| x \|_\infty \) (here \( x_e \) is the exact solution).

Before calling this routine, call the solver routine `?tptrs`.

Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  - Must be 'U' or 'L'.
  - Indicates whether \( A \) is upper or lower triangular:
    - If `uplo` is 'U', then \( A \) is upper triangular.
    - If `uplo` is 'L', then \( A \) is lower triangular.

- **trans**
  - Must be 'N' or 'T' or 'C'.
  - Indicates the form of the equations:
    - If `trans` is 'N', the system has the form \( AX = B \).
    - If `trans` is 'T', the system has the form \( A^T X = B \).
    - If `trans` is 'C', the system has the form \( A^H X = B \).

- **diag**
  - Must be 'N' or 'U'.

If \( \text{diag} = 'N' \), \( A \) is not a unit triangular matrix.

If \( \text{diag} = 'U' \), \( A \) is unit triangular: diagonal elements of \( A \) are assumed to be 1 and not referenced in the array \( \text{ap} \).

\( n \)

The order of the matrix \( A; n \geq 0 \).

\( \text{nrhs} \)

The number of right-hand sides; \( \text{nrhs} \geq 0 \).

\( \text{ap}, \text{b}, \text{x} \)

Arrays:

- \( \text{ap} \) of size \( \text{apmax}(1, n(n + 1)/2) \) contains the upper or lower triangular matrix \( A \), as specified by \( \text{uplo} \).
- \( \text{b} \) of size \( \text{max}(1, \text{ldb} \times \text{nrhs}) \) for column major layout and \( \text{max}(1, \text{ldb} \times n) \) for row major layout contains the right-hand side matrix \( B \).
- \( \text{x} \) of size \( \text{max}(1, \text{ldx} \times \text{nrhs}) \) for column major layout and \( \text{max}(1, \text{ldx} \times n) \) for row major layout contains the solution matrix \( X \).

\( \text{ldb} \)

The leading dimension of \( \text{b} \); \( \text{ldb} \geq \text{max}(1, n) \) for column major layout and \( \text{ldb} \geq \text{nrhs} \) for row major layout.

\( \text{ldx} \)

The leading dimension of \( \text{x} \); \( \text{ldx} \geq \text{max}(1, n) \) for column major layout and \( \text{ldx} \geq \text{nrhs} \) for row major layout.

Output Parameters

\( \text{ferr}, \text{berr} \)

Arrays, size at least \( \text{max}(1, \text{nrhs}) \). Contain the component-wise forward and backward errors, respectively, for each solution vector.

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

Application Notes

The bounds returned in \( \text{ferr} \) are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations \( A^*x = b \); the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately \( n^2 \) floating-point operations for real flavors or \( 4n^2 \) for complex flavors.

See Also

Matrix Storage Schemes

\texttt{tbrfs}

Estimates the error in the solution of a system of linear equations with a triangular band coefficient matrix.

Syntax

\begin{verbatim}
lapack_int LAPACKE_stbrfs( int matrix_layout, char uplo, char trans, char diag, lapack_int n, lapack_int kd, lapack_int nrhs, const float* ab, lapack_int ldab, const float* b, lapack_int ldb, const float* x, lapack_int ldx, float* ferr, float* berr );
\end{verbatim}
lapack_int LAPACKE_dtbrfs( int matrix_layout, char uplo, char trans, char diag,
    lapack_int n, lapack_int kd, lapack_int nrhs, const double* ab, lapack_int ldab, const
    double* b, lapack_int ldb, const double* x, lapack_int ldx, double* ferr, double*
berr);

lapack_int LAPACKE_ctbrfs( int matrix_layout, char uplo, char trans, char diag,
    lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_float* ab,
    lapack_int ldab, const lapack_complex_float* b, lapack_int ldb, const
    lapack_complex_float* x, lapack_int ldx, float* ferr, float* berr);

lapack_int LAPACKE_ztbrfs( int matrix_layout, char uplo, char trans, char diag,
    lapack_int n, lapack_int kd, lapack_int nrhs, const lapack_complex_double* ab,
    lapack_int ldab, const lapack_complex_double* b, lapack_int ldb, const
    lapack_complex_double* x, lapack_int ldx, double* ferr, double* berr);

Include Files

• mkl.h

Description

The routine estimates the errors in the solution to a system of linear equations $A\cdot X = B$ or $A^T\cdot X = B$ or $A^H\cdot X = B$ with a triangular band matrix $A$, with multiple right-hand sides. For each computed solution vector $x$, the routine computes the component-wise backward error. This error is the smallest relative perturbation in elements of $A$ and $b$ such that $x$ is the exact solution of the perturbed system:

$$
|\delta a_{ij}| \leq \beta |a_{ij}|, \quad |\delta b_i| \leq \beta |b_i| \quad \text{such that} \quad (A + \delta A) \cdot x = (b + \delta b).
$$

The routine also estimates the component-wise forward error in the computed solution $||x - x_e||_\infty / ||x||_\infty$ (here $x_e$ is the exact solution).

Before calling this routine, call the solver routine ?tbtrs.

Input Parameters

**matrix_layout**

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**uplo**

Must be 'U' or 'L'.

Indicates whether $A$ is upper or lower triangular:

- If uplo = 'U', then $A$ is upper triangular.
- If uplo = 'L', then $A$ is lower triangular.

**trans**

Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

- If trans = 'N', the system has the form $A\cdot X = B$.
- If trans = 'T', the system has the form $A^T\cdot X = B$.
- If trans = 'C', the system has the form $A^H\cdot X = B$.

**diag**

Must be 'N' or 'U'.

If diag = 'N', $A$ is not a unit triangular matrix.

If diag = 'U', $A$ is unit triangular: diagonal elements of $A$ are assumed to be 1 and not referenced in the array ab.
The order of the matrix $A$; $n \geq 0$.

The number of super-diagonals or sub-diagonals in the matrix $A$; $kd \geq 0$.

The number of right-hand sides; $nrhs \geq 0$.

Arrays:

$ab$ (size max(1, $ldab*\ n$)) contains the upper or lower triangular matrix $A$, as specified by $uplo$, in band storage format.

$b$ of size max(1, $ldb*nrhs$) for column major layout and max(1, $ldb*n$) for row major layout contains the right-hand side matrix $B$.

$x$ of size max(1, $ldx*nrhs$) for column major layout and max(1, $ldx*n$) for row major layout contains the solution matrix $X$.

The leading dimension of the array $ab$; $ldab \geq kd + 1$.

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

The leading dimension of $x$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Arrays, size at least max(1, $nrhs$). Contain the component-wise forward and backward errors, respectively, for each solution vector.

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

The bounds returned in $ferr$ are not rigorous, but in practice they almost always overestimate the actual error.

A call to this routine involves, for each right-hand side, solving a number of systems of linear equations $A^*x = b$; the number of systems is usually 4 or 5 and never more than 11. Each solution requires approximately $2n*kd$ floating-point operations for real flavors or $8n*kd$ operations for complex flavors.

See Also

Matrix Storage Schemes

Matrix Inversion: LAPACK Computational Routines

It is seldom necessary to compute an explicit inverse of a matrix. In particular, do not attempt to solve a system of equations $Ax = b$ by first computing $A^{-1}$ and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see Routines for Solving Systems of Linear Equations); this is more efficient and more accurate.

However, matrix inversion routines are provided for the rare occasions when an explicit inverse matrix is needed.
Computes the inverse of an LU-factored general matrix.

Syntax

```c
lapack_int LAPACKE_sgetri (int matrix_layout, lapack_int n, float * a, lapack_int lda, const lapack_int * ipiv);
```

```c
lapack_int LAPACKE_dgetri (int matrix_layout, lapack_int n, double * a, lapack_int lda, const lapack_int * ipiv);
```

```c
lapack_int LAPACKE_cgetri (int matrix_layout, lapack_int n, lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv);
```

```c
lapack_int LAPACKE_zgetri (int matrix_layout, lapack_int n, lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv);
```

Include Files

- `mkl.h`

Description

The routine computes the inverse \( \text{inv}(A) \) of a general matrix \( A \). Before calling this routine, call \(?getrf\) to factorize \( A \).

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `n` The order of the matrix \( A \); \( n \geq 0 \).
- `a` Array \( a(\text{size max}(1, \text{lda}*n)) \) contains the factorization of the matrix \( A \), as returned by \(?getrf\): \( A = P*L*U \). The second dimension of \( a \) must be at least \( \text{max}(1, n) \).
- `lda` The leading dimension of \( a \); \( lda \geq \text{max}(1, n) \).
- `ipiv` Array, size at least \( \text{max}(1, n) \). The \( ipiv \) array, as returned by \(?getrf\).

Output Parameters

- `a` Overwritten by the \( n \)-by-\( n \) matrix \( \text{inv}(A) \).

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), the \( i \)-th diagonal element of the factor \( U \) is zero, \( U \) is singular, and the inversion could not be completed.
Application Notes

The computed inverse $X$ satisfies the following error bound:

$$|XA - I| \leq c(n)\varepsilon|X|P|L||U|,$$

where $c(n)$ is a modest linear function of $n$; $\varepsilon$ is the machine precision; $I$ denotes the identity matrix; $P$, $L$, and $U$ are the factors of the matrix factorization $A = P*L*U$.

The total number of floating-point operations is approximately $(4/3)n^3$ for real flavors and $(16/3)n^3$ for complex flavors.

See Also
Matrix Storage Schemes

**mkl_?getrinp**
Computes the inverse of an LU-factored general matrix without pivoting.

Syntax

```c
lapack_int LAPACKE_mkl_sgetrinp (int matrix_layout, lapack_int n, float * a, 
lapack_int lda);

lapack_int LAPACKE_mkl_dgetrinp (int matrix_layout, lapack_int n, double * a, 
lapack_int lda);

lapack_int LAPACKE_mkl_cgetrinp (int matrix_layout, lapack_int n, 
lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_mkl_zgetrinp (int matrix_layout, lapack_int n, 
lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a general matrix $A$. Before calling this routine, call **mkl_?getrfnp** to factorize $A$.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **n**
  The order of the matrix $A$; $n \geq 0$.

- **a**
  Array(size max(1, lda*n)) contains the factorization of the matrix $A$, as returned by **mkl_?getrfnp**: $A = L*U$. The second dimension of $a$ must be at least max(1, n).

- **lda**
  The leading dimension of $a$; $lda \geq \text{max}(1, n)$.

Output Parameters

- **a**
  Overwritten by the $n$-by-$n$ matrix $\text{inv}(A)$.
Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, the i-th diagonal element of the factor U is zero, U is singular, and the inversion could not be completed.

Application Notes
The total number of floating-point operations is approximately \((4/3)n^3\) for real flavors and \((16/3)n^3\) for complex flavors.

See Also
Matrix Storage Schemes

?potri
Computes the inverse of a symmetric (Hermitian) positive-definite matrix using the Cholesky factorization.

Syntax
lapack_int LAPACKE_spotri (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda);
lapack_int LAPACKE_dpotri (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda);
lapack_int LAPACKE_cpotri (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda);
lapack_int LAPACKE_zpotri (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda);

Include Files
• mkl.h

Description
The routine computes the inverse \(\text{inv}(A)\) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix \(A\). Before calling this routine, call ?potrf to factorize \(A\).

Input Parameters

matrix_layout
规定的矩阵存储布局是行优先（LAPACK_ROW_MAJOR）还是列优先（LAPACK_COL_MAJOR）。

uplo

Must be 'U' or 'L'.

Indicates how the input matrix \(A\) has been factored:
If uplo = 'U', the upper triangle of \(A\) is stored.
If uplo = 'L', the lower triangle of \(A\) is stored.

The order of the matrix \(A\); \(n \geq 0\).
Array \( a(\text{size max}(1, \text{lda}*n)) \). Contains the factorization of the matrix \( A \), as returned by \( \text{?potrf} \).

\( \text{lda} \)

The leading dimension of \( a \). \( \text{lda} \geq \text{max}(1, n) \).

Output Parameters

\( a \)

Overwritten by the upper or lower triangle of the inverse of \( A \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), the \( i \)-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.

Application Notes

The computed inverse \( X \) satisfies the following error bounds:

\[
||X - I||_2 \leq c(n) \kappa_2(A),
||AX - I||_2 \leq c(n) \kappa_2(A),
\]

where \( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The 2-norm \( ||A||_2 \) of a matrix \( A \) is defined by \( ||A||_2 = \max_{x \neq 0} (Ax \cdot Ax)^{1/2} \), and the condition number \( \kappa_2(A) \) is defined by \( \kappa_2(A) = ||A||_2 ||A^{-1}||_2 \).

The total number of floating-point operations is approximately \((2/3) n^3\) for real flavors and \((8/3) n^3\) for complex flavors.

See Also

Matrix Storage Schemes

\( \text{?pftri} \)

Computes the inverse of a symmetric (Hermitian) positive-definite matrix in RFP format using the Cholesky factorization.

Syntax

```c
lapack_int LAPACK_spftri (int \text{matrix_layout} , char \text{transr} , char \text{uplo} , lapack_int \text{n} , float * a );

lapack_int LAPACK_dpftri (int \text{matrix_layout} , char \text{transr} , char \text{uplo} , lapack_int \text{n} , double * a );

lapack_int LAPACK_cpftri (int \text{matrix_layout} , char \text{transr} , char \text{uplo} , lapack_int \text{n} , lapack_complex_float * a );

lapack_int LAPACK_zpftri (int \text{matrix_layout} , char \text{transr} , char \text{uplo} , lapack_int \text{n} , lapack_complex_double * a );
```

Include Files

- mkl.h

Description
The routine computes the inverse \( \text{inv}(A) \) of a symmetric positive definite or, for complex data, Hermitian positive-definite matrix \( A \) using the Cholesky factorization:

\[
A = U^T U \text{ for real data, } A = U^H U \text{ for complex data} \quad \text{if } \text{uplo} = 'U'
\]
\[
A = L L^T \text{ for real data, } A = L L^H \text{ for complex data} \quad \text{if } \text{uplo} = 'L'
\]

Before calling this routine, call \( \text{?pftrf} \) to factorize \( A \).

The matrix \( A \) is in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **transr**
  Must be 'N', 'T' (for real data) or 'C' (for complex data).

  If \( \text{transr} = 'N' \), the Normal \( \text{transr} \) of RFP \( U \) (if \( \text{uplo} = 'U' \)) or \( L \) (if \( \text{uplo} = 'L' \)) is stored.

  If \( \text{transr} = 'T' \), the Transpose \( \text{transr} \) of RFP \( U \) (if \( \text{uplo} = 'U' \)) or \( L \) (if \( \text{uplo} = 'L' \)) is stored.

  If \( \text{transr} = 'C' \), the Conjugate-Transpose \( \text{transr} \) of RFP \( U \) (if \( \text{uplo} = 'U' \)) or \( L \) (if \( \text{uplo} = 'L' \)) is stored.

- **uplo**
  Must be 'U' or 'L'.

  Indicates how the input matrix \( A \) has been factored:

  If \( \text{uplo} = 'U' \), \( A = U^T U \) for real data or \( A = U^H U \) for complex data, and \( U \) is stored.

  If \( \text{uplo} = 'L' \), \( A = L L^T \) for real data or \( A = L L^H \) for complex data, and \( L \) is stored.

- **n**
  The order of the matrix \( A \); \( n \geq 0 \).

- **a**
  Array, size \( (n \times (n+1)/2) \). The array \( a \) contains the factor \( U \) or \( L \) matrix \( A \) in the RFP format.

**Output Parameters**

- **a**
  The symmetric/Hermitian inverse of the original matrix in the same storage format.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), the \( (i,i) \) element of the factor \( U \) or \( L \) is zero, and the inverse could not be computed.

**See Also**

Matrix Storage Schemes
?pptri
Computes the inverse of a packed symmetric (Hermitian) positive-definite matrix using Cholesky factorization.

Syntax

```c
lapack_int LAPACKE_spptri (int matrix_layout, char uplo, lapack_int n, float * ap);

lapack_int LAPACKE_dpptri (int matrix_layout, char uplo, lapack_int n, double * ap);

lapack_int LAPACKE_cpptri (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * ap);

lapack_int LAPACKE_zpptri (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * ap);
```

Include Files

- mkl.h

Description

The routine computes the inverse \( \text{inv}(A) \) of a symmetric positive definite or, for complex flavors, Hermitian positive-definite matrix \( A \) in packed form. Before calling this routine, call ?pptrf to factorize \( A \).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.

  Indicates whether the upper or lower triangular factor is stored in \( ap \):
  - If \( \text{uplo} = 'U' \), then the upper triangular factor is stored.
  - If \( \text{uplo} = 'L' \), then the lower triangular factor is stored.

- **n**
  The order of the matrix \( A \); \( n \geq 0 \).

- **ap**
  Array, size at least \( \text{max}(1, n(n+1)/2) \).

  Contains the factorization of the packed matrix \( A \), as returned by ?pptrf.

  The dimension \( ap \) must be at least \( \text{max}(1,n(n+1)/2) \).

Output Parameters

- **ap**
  Overwritten by the packed \( n \)-by-\( n \) matrix \( \text{inv}(A) \).

Return Values

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), parameter \( i \) had an illegal value.
- If \( \text{info} = i \), the \( i \)-th diagonal element of the Cholesky factor (and therefore the factor itself) is zero, and the inversion could not be completed.
Application Notes
The computed inverse $X$ satisfies the following error bounds:

\[ ||X - I||_2 \leq c(n) \kappa_2(A), \quad ||AX - I||_2 \leq c(n) \kappa(A), \]

where $c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision; $I$ denotes the identity matrix.

The 2-norm $||A||_2$ of a matrix $A$ is defined by $||A||_2 = \max_{\|x\|=1} (Ax \cdot Ax)^{1/2}$, and the condition number $\kappa_2(A)$ is defined by $\kappa_2(A) = ||A||_2 \cdot ||A^{-1}||_2$.

The total number of floating-point operations is approximately $\frac{2}{3} n^3$ for real flavors and $\frac{8}{3} n^3$ for complex flavors.

See Also
Matrix Storage Schemes

sytri
Computes the inverse of a symmetric matrix using $U^D*U^T$ or $L*D*L^T$ Bunch-Kaufman factorization.

Syntax

```c
lapack_int LAPACKE_ssytri (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda, const lapack_int * ipiv);
lapack_int LAPACKE_dsytri (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda, const lapack_int * ipiv);
lapack_int LAPACKE_csytri (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv);
lapack_int LAPACKE_zsytri (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv);
```

Include Files

- mkl.h

Description
The routine computes the inverse $\text{inv}(A)$ of a symmetric matrix $A$. Before calling this routine, call sytrf to factorize $A$.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be ‘U’ or ‘L’.
- **n**: The order of the matrix $A$; $n \geq 0$. 

$a$

$a$ contains the factorization of the matrix $A$, as returned by _?sytrf_.

$lda$

The leading dimension of $a$; $lda \geq \max(1, n)$.

$ipiv$

Array, size at least $\max(1, n)$.

The $ipiv$ array, as returned by _?sytrf_.

**Output Parameters**

$a$

Overwritten by the $n$-by-$n$ matrix $\text{inv}(A)$.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, the $i$-th diagonal element of $D$ is zero, $D$ is singular, and the inversion could not be completed.

**Application Notes**

The computed inverse $X$ satisfies the following error bounds:

$$ |D^*U^T P^T X^* P^T U - I| \leq c(n) \varepsilon (|D| |U^T| P^T |X| P |U| + |D| |D^{-1}|) $$

for $uplo = 'U'$, and

$$ |D^*L^T P^T X^* P^T L - I| \leq c(n) \varepsilon (|D| |L^T| P^T |X| P |L| + |D| |D^{-1}|) $$

for $uplo = 'L$'. Here $c(n)$ is a modest linear function of $n$, and $\varepsilon$ is the machine precision; $I$ denotes the identity matrix.

The total number of floating-point operations is approximately $\frac{2}{3} n^3$ for real flavors and $\frac{8}{3} n^3$ for complex flavors.

**See Also**

Matrix Storage Schemes

_?hetri_

*Computes the inverse of a complex Hermitian matrix using $U^*D^*U$ or $L^*D^*L^H$ Bunch-Kaufman factorization.*

**Syntax**

```c
lapack_int LAPACKE_chetri (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv);

lapack_int LAPACKE_zhetri (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**
The routine computes the inverse \( \text{inv} (A) \) of a complex Hermitian matrix \( A \). Before calling this routine, call \(?\text{hetrf}\) to factorize \( A \).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major \((\text{LAPACK\_ROW\_MAJOR})\) or column major \((\text{LAPACK\_COL\_MAJOR})\).

- **uplo**
  Must be 'U' or 'L'.
  Indicates how the input matrix \( A \) has been factored:
  - If \( \text{uplo} = 'U' \), the array \( a \) stores the Bunch-Kaufman factorization \( A = U*D*U^H \).
  - If \( \text{uplo} = 'L' \), the array \( a \) stores the Bunch-Kaufman factorization \( A = L*D*L^H \).

- **n**
  The order of the matrix \( A \); \( n \geq 0 \).

- **a**, array \( a(\text{size max}(1, \text{lda}*n)) \)
  Contains the factorization of the matrix \( A \), as returned by \(?\text{hetrf}\). The second dimension of \( a \) must be at least \( \text{max}(1, n) \).

- **lda**
  The leading dimension of \( a \); \( \text{lda} \geq \text{max}(1, n) \).

- **ipiv**
  Array, size at least \( \text{max}(1, n) \). The \( \text{ipiv} \) array, as returned by \(?\text{hetrf}\).

**Output Parameters**

- **a**
  Overwritten by the \( n \)-by-\( n \) matrix \( \text{inv}(A) \).

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), parameter \( i \) had an illegal value.
- If \( \text{info} = i \), the \( i \)-th diagonal element of \( D \) is zero, \( D \) is singular, and the inversion could not be completed.

**Application Notes**

The computed inverse \( X \) satisfies the following error bounds:

\[
|D*U^H*P^T*X*P*U - I| \leq c(n)\varepsilon(\|D\|\|U^H\|^2\|P\|\|X\|\|P\|\|U\| + |D||D^{-1}|)
\]

for \( \text{uplo} = 'U' \), and

\[
|D*L^H*P^T*X*P*L - I| \leq c(n)\varepsilon(\|D\|\|L^H\|^2\|P\|\|X\|\|P\|\|L\| + |D||D^{-1}|)
\]

for \( \text{uplo} = 'L' \). Here \( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The total number of floating-point operations is approximately \( (8/3)n^3 \) for complex flavors.

The real counterpart of this routine is \(?\text{sytri}\).

**See Also**

Matrix Storage Schemes
?sytri2
Computes the inverse of a symmetric indefinite matrix through allocating memory and calling ?sytri2x.

Syntax

```c
lapack_int LAPACKE_ssytri2 (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_dsytri2 (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_csytri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv );

lapack_int LAPACKE_zsytri2 (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv );
```

Include Files

- mkl.h

Description

The routine computes the inverse \( \text{inv}(A) \) of a symmetric indefinite matrix \( A \) using the factorization \( A = U^*D*U \) or \( A = L*D*L^T \) computed by ?sytrf.

The ?sytri2 routine allocates a temporary buffer before calling ?sytri2x that actually computes the inverse.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo` Must be 'U' or 'L'. Indicates how the input matrix \( A \) has been factored:
  - If `uplo` = 'U', the array \( a \) stores the factorization \( A = U^*D*U \).
  - If `uplo` = 'L', the array \( a \) stores the factorization \( A = L*D*L^T \).
- `n` The order of the matrix \( A \); \( n \geq 0 \).
- `a` Array \( a(\text{size max}(1, lda*n)) \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as returned by ?sytrf.
- `lda` The leading dimension of \( a \); \( lda \geq \text{max}(1, n) \).
- `ipiv` Array, size at least \( \text{max}(1, n) \). Details of the interchanges and the block structure of \( D \) as returned by ?sytrf.

Output Parameters

- `a` If `info = 0`, the symmetric inverse of the original matrix.
  - If `uplo` = 'U', the upper triangular part of the inverse is formed and the part of \( A \) below the diagonal is not referenced.
If uplo = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

**Return Values**

This function returns a value info.

- If info = 0, the execution is successful.
- If info = -i, the i-th parameter had an illegal value.
- If info = i, D(i,i) = 0; D is singular and its inversion could not be computed.

**See Also**

- ?sytrf
- ?sytri2x

**Matrix Storage Schemes**

- ?hetri2

  *Computes the inverse of a Hermitian indefinite matrix through allocating memory and calling ?hetri2x.*

**Syntax**

```c
lapack_int LAPACKE_chetri2 (int matrix_layout, char uplo, lapack_int n, 
lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv );

lapack_int LAPACKE_zhetri2 (int matrix_layout, char uplo, lapack_int n, 
lapack_complex_double * a, lapack_int lda, const lapack_int * ipiv );
```

**Include Files**

- mkl.h

**Description**

The routine computes the inverse \( \text{inv}(A) \) of a Hermitian indefinite matrix \( A \) using the factorization \( A = U \cdot D \cdot U^H \) or \( A = L \cdot D \cdot L^H \) computed by ?hetrf.

The ?hetri2 routine allocates a temporary buffer before calling ?hetri2x that actually computes the inverse.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  
  Must be 'U' or 'L'.

  Indicates how the input matrix \( A \) has been factored:

  - If uplo = 'U', the array \( a \) stores the factorization \( A = U \cdot D \cdot U^H \).
  - If uplo = 'L', the array \( a \) stores the factorization \( A = L \cdot D \cdot L^H \).

- **n**
  
  The order of the matrix \( A \); \( n \geq 0 \).

- **a**
  
  Array \( a(\text{size max}(1, lda \cdot n)) \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as returned by ?sytrf.
The leading dimension of $a$; $lda \geq \max(1, n)$.

Array, size at least $\max(1, n)$.

Details of the interchanges and the block structure of $D$ as returned by ?hetrf.

**Output Parameters**

$a$

If $info = 0$, the inverse of the original matrix.

If $uplo = 'U'$, the upper triangular part of the inverse is formed and the part of $A$ below the diagonal is not referenced.

If $uplo = 'L'$, the lower triangular part of the inverse is formed and the part of $A$ above the diagonal is not referenced.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, $D(i,i) = 0$; $D$ is singular and its inversion could not be computed.

**See Also**

?hetrf

?hetri2x

**Matrix Storage Schemes**

?sytri2x

*Comes the inverse of a symmetric indefinite matrix after ?sytri2 allocates memory.*

**Syntax**

```c
lapack_int LAPACKE_ssytri2x (int matrix_layout, char uplo, lapack_int n, float *a, lapack_int lda, const lapack_int *ipiv, lapack_int nb);

lapack_int LAPACKE_dsytri2x (int matrix_layout, char uplo, lapack_int n, double *a, lapack_int lda, const lapack_int *ipiv, lapack_int nb);

lapack_int LAPACKE_csytri2x (int matrix_layout, char uplo, lapack_int n, lapack_complex_float *a, lapack_int lda, const lapack_int *ipiv, lapack_int nb);

lapack_int LAPACKE_zsytri2x (int matrix_layout, char uplo, lapack_int n, lapack_complex_double *a, lapack_int lda, const lapack_int *ipiv, lapack_int nb);
```

**Include Files**

* mkl.h

**Description**

The routine computes the inverse $\text{inv}(A)$ of a symmetric indefinite matrix $A$ using the factorization $A = U^*D*U$ or $A = L^*D*L^*$ computed by ?sytrf.

The ?sytri2x actually computes the inverse after the ?sytri2 routine allocates memory before calling ?sytri2x.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates how the input matrix A has been factored:
  - If uplo = 'U', the array a stores the factorization \( A = U^T D U \).
  - If uplo = 'L', the array a stores the factorization \( A = L D L^T \).

- **n**
  The order of the matrix A; \( n \geq 0 \).

- **a**
  Array \( a \) (size \( \text{max}(1, lda*n) \)) contains the \( nb \) (block size) diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as returned by `?sytrf`. The second dimension of \( a \) must be at least \( \text{max}(1, n) \).
  - The leading dimension of \( a \); \( lda \geq \text{max}(1, n) \).

- **ipiv**
  Array, size at least \( \text{max}(1, n) \).
  Details of the interchanges and the \( nb \) structure of \( D \) as returned by `?sytrf`.

- **nb**
  Block size.

**Output Parameters**

- **a**
  - If \( info = 0 \), the symmetric inverse of the original matrix.
  - If \( info = 'U' \), the upper triangular part of the inverse is formed and the part of \( A \) below the diagonal is not referenced.
  - If \( info = 'L' \), the lower triangular part of the inverse is formed and the part of \( A \) above the diagonal is not referenced.

**Return Values**

This function returns a value \( info \).

- If \( info = 0 \), the execution is successful.
- If \( info = -i \), parameter \( i \) had an illegal value.
- If \( info = i \), \( D_{ii} = 0; D \) is singular and its inversion could not be computed.

**See Also**

- `?sytrf`
- `?sytri2`

**Matrix Storage Schemes**

- `?hetri2x`
  Computes the inverse of a Hermitian indefinite matrix after `?hetri2allocates memory`. 
Syntax

```c
lapack_int LAPACKE_chetri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

```c
lapack_int LAPACKE_zhetri2x (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda , const lapack_int * ipiv , lapack_int nb );
```

Include Files

- mkl.h

Description

The routine computes the inverse $\text{inv}(A)$ of a Hermitian indefinite matrix $A$ using the factorization $A = U^*D*U$ or $A = L*D*L^H$ computed by ?hetrf.

The ?hetri2x actually computes the inverse after the ?hetri2 routine allocates memory before calling ?hetri2x.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**
  Must be 'U' or 'L'.

  Indicates how the input matrix $A$ has been factored:
  - If $\text{uplo} = 'U'$, the array $a$ stores the factorization $A = U^*D*U^H$.
  - If $\text{uplo} = 'L'$, the array $a$ stores the factorization $A = L*D*L^H$.
- **n**
  The order of the matrix $A$; $n \geq 0$.
- **a**
  Arrays $a(\text{size max}(1, \text{lda} \times n))$ contains the $nb$ (block size) diagonal matrix $D$ and the multipliers used to obtain the factor $U$ or $L$ as returned by ?hetrf.
- **lda**
  The leading dimension of $a$; $\text{lda} \geq \text{max}(1, n)$.
- **ipiv**
  Array, size at least $\text{max}(1, n)$. Details of the interchanges and the $nb$ structure of $D$ as returned by ?hetrf.
- **nb**
  Block size.

Output Parameters

- **a**
  - If $\text{info} = 0$, the symmetric inverse of the original matrix.
  - If $\text{info} = 'U'$, the upper triangular part of the inverse is formed and the part of $A$ below the diagonal is not referenced.
  - If $\text{info} = 'L'$, the lower triangular part of the inverse is formed and the part of $A$ above the diagonal is not referenced.

Return Values

This function returns a value $\text{info}$.
If $\text{info} = 0$, the execution is successful.
If $\text{info} = -i$, parameter $i$ had an illegal value.
If $\text{info} = i$, $D_{ii} = 0$; $D$ is singular and its inversion could not be computed.

See Also
hetrf
hetri2
Matrix Storage Schemes

?sytri_3
Computes the inverse of a real or complex symmetric matrix.

```c
lapack_int LAPACKE_ssytri_3 (int matrix_layout, char uplo, lapack_int n, float * A,
lapack_int lda, const float * e, const lapack_int * ipiv);
lapack_int LAPACKE_dsytri_3 (int matrix_layout, char uplo, lapack_int n, double * A,
lapack_int lda, const double * e, const lapack_int * ipiv);
lapack_int LAPACKE_csytri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float * A, lapack_int lda, const lapack_complex_float * e, const
lapack_int * ipiv);
lapack_int LAPACKE_zsytri_3 (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double * A, lapack_int lda, const lapack_complex_double * e, const
lapack_int * ipiv);
```

Description

?sytri_3 computes the inverse of a real or complex symmetric matrix $A$ using the factorization computed by ?sytrf_rk: $A = P*U*D*(U^T)*(P^T)$ or $A = P*L*D*(L^T)*(P^T)$, where $U$ (or $L$) is a unit upper (or lower) triangular matrix, $U^T$ (or $L^T$) is the transpose of $U$ (or $L$), $P$ is a permutation matrix, $P^T$ is the transpose of $P$, and $D$ is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?sytri_3 sets the leading dimension of the workspace before calling ?sytri_3x, which actually computes the inverse. This is the blocked version of the algorithm, calling Level-3 BLAS.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.
  - = 'U': The upper triangle of $A$ is stored.
  - = 'L': The lower triangle of $A$ is stored.
- **n**: The order of the matrix $A$. $n \geq 0$.
- **A**: Array of size max(1, $\text{lda} \times n$). On entry, diagonal of the block diagonal matrix $D$ and factors $U$ or $L$ as computed by ?sytrf_rk:
  - Only diagonal elements of the symmetric block diagonal matrix $D$ on the diagonal of $A$; that is, $D(k,k) = A(k,k)$. Superdiagonal (or subdiagonal) elements of $D$ should be provided on entry in array $e$. —and—
• If uplo = 'U', factor U in the superdiagonal part of A. If uplo = 'L', factor L in the subdiagonal part of A.

lda
The leading dimension of the array A.

e
Array of size n. On entry, contains the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If uplo = 'U', e(i) = D(i-1:i), i=2:N, and e(1) is not referenced. If uplo = 'L', e(i) = D(i:i+1), i=1:N-1, and e(n) is not referenced.

NOTE For 1-by-1 diagonal block D(k), where 1 ≤ k ≤ n, the element e[k-1] is not referenced in both the uplo = 'U' and uplo = 'L' cases.

ipiv
Array of size n. Details of the interchanges and the block structure of D as determined by ?sytrf_rk.

Output Parameters

A
On exit, if info = 0, the symmetric inverse of the original matrix. If uplo = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If uplo = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

Return Values

This function returns a value info.

= 0: Successful exit.
< 0: If info = -i, the i-th argument had an illegal value.
> 0: If info = i, D(i,i) = 0; the matrix is singular and its inverse could not be computed.

?hetri_3
Computes the inverse of a complex Hermitian matrix using the factorization computed by ?hetrf_rk.

?hetri_3 computes the inverse of a complex Hermitian matrix A using the factorization computed by ?hetrf_rk: A = P*U*D*(U^H)*(P^T) or A = P*L*D*(L^H)*(P^T), where U (or L) is a unit upper (or lower) triangular matrix, U^H (or L^H) is the conjugate of U (or L), P is a permutation matrix, P^T is the transpose of P, and D is a Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetri_3 sets the leading dimension of the workspace before calling ?hetri_3x, which actually computes the inverse.
This is the blocked version of the algorithm, calling Level-3 BLAS.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **uplo**: Specifies whether the details of the factorization are stored as an upper or lower triangular matrix.
  - = 'U': The upper triangle of A is stored.
  - = 'L': The lower triangle of A is stored.

- **n**: The order of the matrix A. \( n \geq 0 \).

- **A**: Array of size \( \max(1, lda*n) \). On entry, diagonal of the block diagonal matrix D and factor U or L as computed by \(?hetrf_rk\):
  - Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, \( D(k,k) = A(k,k) \). Superdiagonal (or subdiagonal) elements of D should be provided on entry in array \( e \).
  - If \( uplo = 'U' \), factor U in the superdiagonal part of A. If \( uplo = 'L' \), factor L is the subdiagonal part of A.

- **lda**: The leading dimension of the array A.

- **e**: Array of size \( n \). On entry, contains the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If \( uplo = 'U' \), \( e(i) = D(i-1,i) \), \( i=2:N \), and \( e(1) \) is not referenced. If \( uplo = 'L' \), \( e(i) = D(i+1,i) \), \( i=1:N-1 \), and \( e(n) \) is not referenced.

  **NOTE**: For 1-by-1 diagonal block \( D(k) \), where \( 1 \leq k \leq n \), the element \( e[k-1] \) is not referenced in both the \( uplo = 'U' \) and \( uplo = 'L' \) cases.

- **ipiv**: Array of size \( n \). Details of the interchanges and the block structure of D as determined by \(?hetrf_rk\).

**Output Parameters**

- **A**: On exit, if \( info = 0 \), the Hermitian inverse of the original matrix. If \( uplo = 'U' \), the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced. If \( uplo = 'L' \), the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

**Return Values**

This function returns a value \( info \).

- \( = 0 \): Successful exit.
- \( < 0 \): If \( info = -i \), the \( i^{th} \) argument had an illegal value.
- \( > 0 \): If \( info = i \), \( D(i,i) = 0 \); the matrix is singular and its inverse could not be computed.
Computes the inverse of a symmetric matrix using \(U^\star D \star U^T\) or \(L^\star D \star L^T\) Bunch-Kaufman factorization of matrix in packed storage.

**Syntax**

```c
lapack_int LAPACKE_ssptri (int matrix_layout, char uplo, lapack_int n, float * ap, const lapack_int * ipiv);
lapack_int LAPACKE_dsptri (int matrix_layout, char uplo, lapack_int n, double * ap, const lapack_int * ipiv);
lapack_int LAPACKE_csptri (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * ap, const lapack_int * ipiv);
lapack_int LAPACKE_zsptri (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * ap, const lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**

The routine computes the inverse \(\text{inv}(A)\) of a packed symmetric matrix \(A\). Before calling this routine, call \(?\text{sptrf}\) to factorize \(A\).

**Input Parameters**

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `uplo`
  Must be 'U' or 'L'.
  Indicates how the input matrix \(A\) has been factored:
  - If `uplo` = 'U', the array `ap` stores the Bunch-Kaufman factorization \(A = U^\star D \star U^T\).
  - If `uplo` = 'L', the array `ap` stores the Bunch-Kaufman factorization \(A = L^\star D \star L^T\).

- `n`
  The order of the matrix \(A\); \(n \geq 0\).

- `ap`
  Arrays `ap` (size \(\max(1, n(n+1)/2)\)) contains the factorization of the matrix \(A\), as returned by \(?\text{sptrf}\).

- `ipiv`
  Array, size at least \(\max(1, n)\). The `ipiv` array, as returned by \(?\text{sptrf}\).

**Output Parameters**

- `ap`
  Overwritten by the matrix \(\text{inv}(A)\) in packed form.

**Return Values**

This function returns a value `info`.

- If `info` = 0, the execution is successful.
- If `info` = \(-i\), parameter \(i\) had an illegal value.
If \( info = i \), the \( i \)-th diagonal element of \( D \) is zero, \( D \) is singular, and the inversion could not be completed.

**Application Notes**

The computed inverse \( X \) satisfies the following error bounds:

\[
|D^*U^*P^*X*P*U - I| \leq c(n)\varepsilon(|D| |U^*P^*X| |P|U| + |D| |D^{-1}|)
\]

for \( uplo = 'U' \), and

\[
|D^*L^*P^*X*P*L - I| \leq c(n)\varepsilon(|D| |L^*P^*X| |P|L| + |D| |D^{-1}|)
\]

for \( uplo = 'L' \). Here \( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The total number of floating-point operations is approximately \((2/3)n^3\) for real flavors and \((8/3)n^3\) for complex flavors.

**See Also**

Matrix Storage Schemes

)?hptri

*Computes the inverse of a complex Hermitian matrix using \( U^*D^*U \) or \( L^*D^*L^H \) Bunch-Kaufman factorization of matrix in packed storage.*

**Syntax**

```c
lapack_int LAPACKE_chptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * ap , const lapack_int * ipiv);
```

```c
lapack_int LAPACKE_zhptri (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * ap , const lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**

The routine computes the inverse \( \text{inv}(A) \) of a complex Hermitian matrix \( A \) using packed storage. Before calling this routine, call ?hptrf to factorize \( A \).

**Input Parameters**

- \( matrix\_layout \): Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \( uplo \): Must be \('U'\) or \('L'\).
  Indicates how the input matrix \( A \) has been factored:
  - If \( uplo = 'U' \), the array \( ap \) stores the packed Bunch-Kaufman factorization \( A = U^*D^*U^T \).
  - If \( uplo = 'L' \), the array \( ap \) stores the packed Bunch-Kaufman factorization \( A = L^*D^*L^T \).
- \( n \): The order of the matrix \( A \); \( n \geq 0 \).
Array \( ap \) (size \( \max(1,n(n+1)/2) \)) contains the factorization of the matrix \( A \), as returned by \(?hptrf\).

Array, size at least \( \max(1, n) \).

The \( ipiv \) array, as returned by \(?hptrf\).

**Output Parameters**

\( ap \) Overwritten by the matrix \( \text{inv}(A) \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), the \( i \)-th diagonal element of \( D \) is zero, \( D \) is singular, and the inversion could not be completed.

**Application Notes**

The computed inverse \( X \) satisfies the following error bounds:

\[
| D^*U^*P^*X^*P^*U - I | \leq c(n)\varepsilon \left( |D| |U^*P^*X|P|U| + |D| |D^{-1}| \right)
\]

for \( \text{uplo} = 'U' \), and

\[
| D^*L^*P^*X^*PL - I | \leq c(n)\varepsilon \left( |D^*L^*P^*X|P|L| + |D| |D^{-1}| \right)
\]

for \( \text{uplo} = 'L' \). Here \( c(n) \) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The total number of floating-point operations is approximately \( (8/3)n^3 \).

The real counterpart of this routine is \(?sptri\).

**See Also**

Matrix Storage Schemes

\(?trtri\)

* Computes the inverse of a triangular matrix.

**Syntax**

```c
lapack_int LAPACKE_strtri (int matrix_layout, char uplo, char diag, lapack_int n, float * a, lapack_int lda);

lapack_int LAPACKE_dtrtri (int matrix_layout, char uplo, char diag, lapack_int n, double * a, lapack_int lda);

lapack_int LAPACKE_ctrtri (int matrix_layout, char uplo, char diag, lapack_int n, lapack_complex_float * a, lapack_int lda);

lapack_int LAPACKE_ztrtri (int matrix_layout, char uplo, char diag, lapack_int n, lapack_complex_double * a, lapack_int lda);
```

**Include Files**

- mkl.h
Description
The routine computes the inverse \( \text{inv}(A) \) of a triangular matrix \( A \).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether \( A \) is upper or lower triangular:
  - If \( \text{uplo} = 'U' \), then \( A \) is upper triangular.
  - If \( \text{uplo} = 'L' \), then \( A \) is lower triangular.

- **diag**
  Must be 'N' or 'U'.
  - If \( \text{diag} = 'N' \), then \( A \) is not a unit triangular matrix.
  - If \( \text{diag} = 'U' \), \( A \) is unit triangular: diagonal elements of \( A \) are assumed to be 1 and not referenced in the array \( a \).

- **n**
  The order of the matrix \( A \); \( n \geq 0 \).

- **a**
  Array: . Contains the matrix \( A \).

- **lda**
  The first dimension of \( a \); \( \text{lda} \geq \max(1, n) \).

**Output Parameters**

- **a**
  Overwritten by the matrix \( \text{inv}(A) \).

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), parameter \( i \) had an illegal value.
- If \( \text{info} = i \), the \( i \)-th diagonal element of \( A \) is zero, \( A \) is singular, and the inversion could not be completed.

**Application Notes**

The computed inverse \( X \) satisfies the following error bounds:

\[
|XA - I| \leq c(n)\varepsilon |X| |A|
\]

\[
|XA - I| \leq c(n)\varepsilon |A^{-1}| |A| |X|,
\]

where \( c(n) \) is a modest linear function of \( n \); \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The total number of floating-point operations is approximately \( (1/3)n^3 \) for real flavors and \( (4/3)n^3 \) for complex flavors.

**See Also**

Matrix Storage Schemes
Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format.

**Syntax**

```c
lapack_int LAPACKE_stftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , float * a );

lapack_int LAPACKE_dtftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , double * a );

lapack_int LAPACKE_ctftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , lapack_complex_float * a );

lapack_int LAPACKE_ztftri (int matrix_layout , char transr , char uplo , char diag ,
lapack_int n , lapack_complex_double * a );
```

**Include Files**
- mkl.h

**Description**

Computes the inverse of a triangular matrix stored in the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

This is the block version of the algorithm, calling Level 3 BLAS.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **transr**
  Must be 'N', 'T' (for real data) or 'C' (for complex data).
  If `transr` = 'N', the Normal `transr` of RFP A is stored.
  If `transr` = 'T', the Transpose `transr` of RFP A is stored.
  If `transr` = 'C', the Conjugate-Transpose `transr` of RFP A is stored.

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of RFP A is stored:
  If `uplo` = 'U', the array `a` stores the upper triangular part of the matrix A.
  If `uplo` = 'L', the array `a` stores the lower triangular part of the matrix A.

- **diag**
  Must be 'N' or 'U'.
  If `diag` = 'N', then A is not a unit triangular matrix.
  If `diag` = 'U', A is unit triangular: diagonal elements of A are assumed to be 1 and not referenced in the array `a`.

- **n**
  The order of the matrix A; \( n \geq 0 \).
Array, size max(1, n*(n + 1)/2). The array \( a \) contains the matrix \( A \) in the RFP format.

**Output Parameters**

\( a \)  
The (triangular) inverse of the original matrix in the same storage format.

**Return Values**

This function returns a value \( info \).
- If \( info=0 \), the execution is successful.
- If \( info = -i \), parameter \( i \) had an illegal value.
- If \( info = i, A_{i, i} \) is exactly zero. The triangular matrix is singular and its inverse cannot be computed.

**See Also**

Matrix Storage Schemes

?tptri  
*Computes the inverse of a triangular matrix using packed storage.*

**Syntax**

```c
lapack_int LAPACKE_stptri (int matrix_layout, char uplo, char diag, lapack_int n, float * ap);
lapack_int LAPACKE_dtptri (int matrix_layout, char uplo, char diag, lapack_int n, double * ap);
lapack_int LAPACKE_ctptri (int matrix_layout, char uplo, char diag, lapack_int n, lapack_complex_float * ap);
lapack_int LAPACKE_ztptri (int matrix_layout, char uplo, char diag, lapack_int n, lapack_complex_double * ap);
```

**Include Files**

- mkl.h

**Description**

The routine computes the inverse \( \text{inv}(A) \) of a packed triangular matrix \( A \).

**Input Parameters**

- **matrix_layout**  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**  
  Must be 'U' or 'L'.

  Indicates whether \( A \) is upper or lower triangular:
  - If \( uplo = 'U' \), then \( A \) is upper triangular.
  - If \( uplo = 'L' \), then \( A \) is lower triangular.

- **diag**  
  Must be 'N' or 'U'.

If \( \text{diag} = 'N' \), then \( A \) is not a unit triangular matrix.

If \( \text{diag} = 'U' \), \( A \) is unit triangular: diagonal elements of \( A \) are assumed to be 1 and not referenced in the array \( ap \).

\[ n \]

The order of the matrix \( A \); \( n \geq 0 \).

\[ ap \]

Array, size at least \( \max(1, n(n+1)/2) \).

Contains the packed triangular matrix \( A \).

Output Parameters

\[ ap \]

Overwritten by the packed \( n \)-by-\( n \) matrix \( \text{inv}(A) \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), the \( i \)-th diagonal element of \( A \) is zero, \( A \) is singular, and the inversion could not be completed.

Application Notes

The computed inverse \( X \) satisfies the following error bounds:

\[ |XA - I| \leq c(n)\varepsilon |X||A| \]

\[ |X - A^{-1}| \leq c(n)\varepsilon |A^{-1}||A||X| \]

where \( c(n) \) is a modest linear function of \( n \); \( \varepsilon \) is the machine precision; \( I \) denotes the identity matrix.

The total number of floating-point operations is approximately \( (1/3)n^3 \) for real flavors and \( (4/3)n^3 \) for complex flavors.

See Also

Matrix Storage Schemes

Matrix Equilibration: LAPACK Computational Routines

Routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

?geequ

Computes row and column scaling factors intended to equilibrate a general matrix and reduce its condition number.

Syntax

```c
lapack_int LAPACKE_sgeequ( int matrix_layout, lapack_int m, lapack_int n, const float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```c
lapack_int LAPACKE_dgeequ( int matrix_layout, lapack_int m, lapack_int n, const double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
```

```c
lapack_int LAPACKE_cgeequ( int matrix_layout, lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```c
lapack_int LAPACKE_zgeequ( int matrix_layout, lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
```
lapack_int LAPACKE_zgeequ( int matrix_layout, lapack_int m, lapack_int n, const
lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );

**Include Files**

- mkl.h

**Description**

The routine computes row and column scalings intended to equilibrate an $m$-by-$n$ matrix $A$ and reduce its condition number. The output array $r$ returns the row scale factors and the array $c$ the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix $B$ with elements $b_{ij} = r_{i-1} \cdot a_{ij} \cdot c_{j-1}$ have absolute value 1.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **m**
  The number of rows of the matrix $A$; $m \geq 0$.
- **n**
  The number of columns of the matrix $A$; $n \geq 0$.
- **a**
  Array: size max(1, $lda \cdot n$) for column major layout and max(1, $lda \cdot m$) for row major layout.
  Contains the $m$-by-$n$ matrix $A$ whose equilibration factors are to be computed.
- **lda**
  The leading dimension of $a$; $lda \geq \max(1, m)$.

**Output Parameters**

- **r**, **c**
  Arrays: $r$ (size $m$), $c$ (size $n$).
  If $info = 0$, or $info > m$, the array $r$ contains the row scale factors of the matrix $A$.
  If $info = 0$, the array $c$ contains the column scale factors of the matrix $A$.
- **rowcnd**
  If $info = 0$ or $info > m$, $rowcnd$ contains the ratio of the smallest $r[i]$ to the largest $r[i]$.
- **colcnd**
  If $info = 0$, $colcnd$ contains the ratio of the smallest $c[i]$ to the largest $c[i]$.
- **amax**
  Absolute value of the largest element of the matrix $A$.

**Return Values**

This function returns a value $info$.
If $info = 0$, the execution is successful.
If $info = -i$, parameter $i$ had an illegal value.
If $info = i$, $i > 0$, and $i \leq m$, the $i$-th row of $A$ is exactly zero;
If \( i > m \), the \((i-m)\)th column of \( A \) is exactly zero.

**Application Notes**

All the components of \( r \) and \( c \) are restricted to be between \( \text{SMLNUM} = \text{smallest safe number} \) and \( \text{BIGNUM} = \text{largest safe number} \). Use of these scaling factors is not guaranteed to reduce the condition number of \( A \) but works well in practice.

\( \text{SMLNUM} \) and \( \text{BIGNUM} \) are parameters representing machine precision. You can use the \texttt{?lamch} routines to compute them. For example, compute single precision values of \( \text{SMLNUM} \) and \( \text{BIGNUM} \) as follows:

\[
\text{SMLNUM} = \text{slamch ('s')}
\text{BIGNUM} = \frac{1}{\text{SMLNUM}}
\]

If \( \text{rowcnd} \geq 0.1 \) and \( \text{amax} \) is neither too large nor too small, it is not worth scaling by \( r \).

If \( \text{colcnd} \geq 0.1 \), it is not worth scaling by \( c \).

If \( \text{amax} \) is very close to \( \text{SMLNUM} \) or very close to \( \text{BIGNUM} \), the matrix \( A \) should be scaled.

**See Also**

Error Analysis
Matrix Storage Schemes

\texttt{?geequb}

*Computes row and column scaling factors restricted to a power of radix to equilibrate a general matrix and reduce its condition number.*

**Syntax**

```c
lapack_int LAPACK_sgeequb( int matrix_layout, lapack_int m, lapack_int n, const float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```c
lapack_int LAPACK_dgeequb( int matrix_layout, lapack_int m, lapack_int n, const double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
```

```c
lapack_int LAPACK_cgeequb( int matrix_layout, lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, float* r, float* c, float* rowcnd, float* colcnd, float* amax );
```

```c
lapack_int LAPACK_zgeequb( int matrix_layout, lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, double* r, double* c, double* rowcnd, double* colcnd, double* amax );
```

**Include Files**

- \texttt{mkl.h}

**Description**

The routine computes row and column scalings intended to equilibrate an \( m \)-by-\( n \) general matrix \( A \) and reduce its condition number. The output array \( r \) returns the row scale factors and the array \( c \) - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix \( B \) with elements \( b_{i,j} = r_{[i-1]}a_{i,j}c_{[j-1]} \) have an absolute value of at most the radix.

\( r_{[i-1]} \) and \( c_{[j-1]} \) are restricted to be a power of the radix between \( \text{SMLNUM} = \text{smallest safe number} \) and \( \text{BIGNUM} = \text{largest safe number} \). Use of these scaling factors is not guaranteed to reduce the condition number of \( A \) but works well in practice.
**SMLNUM** and **BIGNUM** are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of **SMLNUM** and **BIGNUM** as follows:

```
SMLNUM = slamch ('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from ?geequ by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between \(\sqrt{\text{radix}}\) and \(1/\sqrt{\text{radix}}\).

**Input Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matrix_layout</code></td>
<td>Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).</td>
</tr>
<tr>
<td><code>m</code></td>
<td>The number of rows of the matrix (A); (m \geq 0).</td>
</tr>
<tr>
<td><code>n</code></td>
<td>The number of columns of the matrix (A); (n \geq 0).</td>
</tr>
<tr>
<td><code>a</code></td>
<td>Array: size max(1, (lda \times n)) for column major layout and max(1, (lda \times m)) for row major layout. Contains the (m)-by-(n) matrix (A) whose equilibration factors are to be computed.</td>
</tr>
<tr>
<td><code>lda</code></td>
<td>The leading dimension of (a); (lda \geq \max(1, m)).</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code>, <code>c</code></td>
<td>Arrays: (r(m), c(n)). If (info = 0), or (info &gt; m), the array (r) contains the row scale factors for the matrix (A). If (info = 0), the array (c) contains the column scale factors for the matrix (A).</td>
</tr>
<tr>
<td><code>rowcnd</code></td>
<td>If (info = 0) or (info &gt; m), (rowcnd) contains the ratio of the smallest (r[i]) to the largest (r[i]). If (rowcnd \geq 0.1), and (amax) is neither too large nor too small, it is not worth scaling by (r).</td>
</tr>
<tr>
<td><code>colcnd</code></td>
<td>If (info = 0), (colcnd) contains the ratio of the smallest (c[i]) to the largest (c[i]). If (colcnd \geq 0.1), it is not worth scaling by (c).</td>
</tr>
<tr>
<td><code>amax</code></td>
<td>Absolute value of the largest element of the matrix (A). If (amax) is very close to (SMLNUM) or very close to (BIGNUM), the matrix should be scaled.</td>
</tr>
</tbody>
</table>

**Return Values**

This function returns a value \(info\).

If \(info = 0\), the execution is successful.

If \(info = -i\), parameter \(i\) had an illegal value.

If \(info = i, i > 0\), and \(i \leq m\), the \(i\)-th row of \(A\) is exactly zero;

\(i > m\), the \((i-m)\)-th column of \(A\) is exactly zero.
See Also
Error Analysis
Matrix Storage Schemes

?gbequ
* Computes row and column scaling factors intended to equilibrate a banded matrix and reduce its condition number.

Syntax

```
lapack_int LAPACKE_sgbequ (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float* rowcnd, float* colcnd, float* amax);
lapack_int LAPACKE_dgbequ (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double* rowcnd, double* colcnd, double* amax);
lapack_int LAPACKE_cgbequ (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float* c, float* rowcnd, float* colcnd, float* amax);
lapack_int LAPACKE_zgbequ (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r, double* c, double* rowcnd, double* colcnd, double* amax);
```

Include Files

```
• mkl.h
```

Description

The routine computes row and column scalings intended to equilibrate an m-by-n band matrix A and reduce its condition number. The output array r returns the row scale factors and the array c the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix B with elements \( b_{ij} = r[i - 1] \cdot a_{ij} \cdot c[j - 1] \) have absolute value 1.

Input Parameters

```
matrix_layout
   Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

m
   The number of rows of the matrix A; n ≥ 0.

n
   The number of columns of the matrix A; n ≥ 0.

kl
   The number of subdiagonals within the band of A; kl ≥ 0.

ku
   The number of superdiagonals within the band of A; ku ≥ 0.

ab
   Array, size max(1, ldab*n) for column major layout and max(1, ldab*m) for row major layout. Contains the original band matrix A.

ldab
   The leading dimension of ab; ldab ≥ kl+ku+1.
```
Output Parameters

$r, c$

Arrays: $r$ (size $m$), $c$ (size $n$).

If $info = 0$, or $info > m$, the array $r$ contains the row scale factors of the matrix $A$.

If $info = 0$, the array $c$ contains the column scale factors of the matrix $A$.

$\text{rowcnd}$

If $info = 0$ or $info > m$, $\text{rowcnd}$ contains the ratio of the smallest $r[i]$ to the largest $r[i]$.

$\text{colcnd}$

If $info = 0$, $\text{colcnd}$ contains the ratio of the smallest $c[i]$ to the largest $c[i]$.

$\text{amax}$

Absolute value of the largest element of the matrix $A$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$ and

- $i \leq m$, the $i$-th row of $A$ is exactly zero;
- $i > m$, the $(i-m)$th column of $A$ is exactly zero.

Application Notes

All the components of $r$ and $c$ are restricted to be between $SMLNUM = \text{smallest safe number}$ and $BIGNUM = \text{largest safe number}$.

Use of these scaling factors is not guaranteed to reduce the condition number of $A$ but works well in practice.

$SMLNUM$ and $BIGNUM$ are parameters representing machine precision. You can use the $?\text{lamch}$ routines to compute them. For example, compute single precision values of $SMLNUM$ and $BIGNUM$ as follows:

```plaintext
SMLNUM = slamch('s')
BIGNUM = 1 / SMLNUM
```

If $\text{rowcnd} \geq 0.1$ and $\text{amax}$ is neither too large nor too small, it is not worth scaling by $r$.

If $\text{colcnd} \geq 0.1$, it is not worth scaling by $c$.

If $\text{amax}$ is very close to $SMLNUM$ or very close to $BIGNUM$, the matrix $A$ should be scaled.

See Also

Error Analysis
Matrix Storage Schemes

$?\text{gbequ}$

Computes row and column scaling factors restricted to a power of radix to equilibrate a banded matrix and reduce its condition number.
Syntax

```c
lapack_int LAPACKE_sgbequb(int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const float* ab, lapack_int ldab, float* r, float* c, float* rowcnd, float* colcnd, float* amax);

lapack_int LAPACKE_dgbequb(int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const double* ab, lapack_int ldab, double* r, double* c, double* rowcnd, double* colcnd, double* amax);

lapack_int LAPACKE_cgbequb(int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_float* ab, lapack_int ldab, float* r, float* c, float* rowcnd, float* colcnd, float* amax);

lapack_int LAPACKE_zgbequb(int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const lapack_complex_double* ab, lapack_int ldab, double* r, double* c, double* rowcnd, double* colcnd, double* amax);
```

Include Files

- `mkl.h`

Description

The routine computes row and column scalings intended to equilibrate an \( m \)-by-\( n \) banded matrix \( A \) and reduce its condition number. The output array \( r \) returns the row scale factors and the array \( c \) - the column scale factors. These factors are chosen to try to make the largest element in each row and column of the matrix \( B \) with elements \( b_{i,j} = r_{[i-1]} \times a_{i,j} \times c_{[j-1]} \) have an absolute value of at most the radix. \( r_{[i]} \) and \( c_{[j]} \) are restricted to be a power of the radix between \( \text{SMLNUM} = \text{smallest safe number} \) and \( \text{BIGNUM} = \text{largest safe number} \). Use of these scaling factors is not guaranteed to reduce the condition number of \( A \) but works well in practice.

\( \text{SMLNUM} \) and \( \text{BIGNUM} \) are parameters representing machine precision. You can use the \(?\text{lamch}\) routines to compute them. For example, compute single precision values of \( \text{SMLNUM} \) and \( \text{BIGNUM} \) as follows:

```c
SMLNUM = slamch('s')
BIGNUM = 1 / SMLNUM
```

This routine differs from \(?\text{gbequ}\) by restricting the scaling factors to a power of the radix. Except for over- and underflow, scaling by these factors introduces no additional rounding errors. However, the scaled entries' magnitudes are no longer equal to approximately 1 but lie between \( \sqrt{\text{radix}} \) and \( 1/\sqrt{\text{radix}} \).

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `m` The number of rows of the matrix \( A \); \( m \geq 0 \).
- `n` The number of columns of the matrix \( A \); \( n \geq 0 \).
- `kl` The number of subdiagonals within the band of \( A \); \( kl \geq 0 \).
- `ku` The number of superdiagonals within the band of \( A \); \( ku \geq 0 \).
- `ab` Array: size max(1, \( \text{ldab} \times n \)) for column major layout and max(1, \( \text{ldab} \times m \)) for row major layout.
- `ldab` The leading dimension of \( A \); \( \text{ldab} \geq \max(1, m) \).
Output Parameters

$r, c$
Arrays: $r$ (size $m$), $c$ (size $n$).
If $info = 0$, or $info > m$, the array $r$ contains the row scale factors for the matrix $A$.
If $info = 0$, the array $c$ contains the column scale factors for the matrix $A$.

$\text{rowcnd}$
If $info = 0$ or $info > m$, $\text{rowcnd}$ contains the ratio of the smallest $r(i)$ to the largest $r(i)$. If $\text{rowcnd} \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by $r$.

$\text{colcnd}$
If $info = 0$, $\text{colcnd}$ contains the ratio of the smallest $c[i]$ to the largest $c[i]$. If $\text{colcnd} \geq 0.1$, it is not worth scaling by $c$.

$\text{amax}$
Absolute value of the largest element of the matrix $A$. If $\text{amax}$ is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values
This function returns a value $info$.
If $info = 0$, the execution is successful.
If $info = -i$, parameter $i$ had an illegal value.
If $info = i$, the $i$-th diagonal element of $A$ is nonpositive.
$i \leq m$, the $i$-th row of $A$ is exactly zero;
$i > m$, the $(i-m)$-th column of $A$ is exactly zero.

See Also
Error Analysis
Matrix Storage Schemes

?poequ
Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```c
lapack_int LAPACKE_spoequ( int matrix_layout, lapack_int n, const float* a, lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_dpoequ( int matrix_layout, lapack_int n, const double* a, lapack_int lda, double* s, double* scond, double* amax );
lapack_int LAPACKE_cpoequ( int matrix_layout, lapack_int n, const lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
lapack_int LAPACKE_zpoequ( int matrix_layout, lapack_int n, const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files
- mkl.h
Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix $A$ and reduce its condition number (with respect to the two-norm). The output array $s$ returns scale factors such that contains

$$1 / \sqrt{a_{i,i}}$$

These factors are chosen so that the scaled matrix $B$ with elements $B_{i,j} = s[i-1] * A_{i,j} * s[j-1]$ has diagonal elements equal to 1.

This choice of $s$ puts the condition number of $B$ within a factor $n$ of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **n**
  The order of the matrix $A$; $n \geq 0$.

- **a**
  Array: size max(1, $lda * n$).
  Contains the $n$-by-$n$ symmetric or Hermitian positive definite matrix $A$ whose scaling factors are to be computed. Only the diagonal elements of $A$ are referenced.

- **lda**
  The leading dimension of $a$; $lda \geq max(1, n)$.

Output Parameters

- **s**
  Array, size $n$.
  If $info = 0$, the array $s$ contains the scale factors for $A$.

- **scond**
  If $info = 0$, $scond$ contains the ratio of the smallest $s[i]$ to the largest $s[i]$.

- **amax**
  Absolute value of the largest element of the matrix $A$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, the $i$-th diagonal element of $A$ is nonpositive.

Application Notes

If $scond \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by $s$.

If $amax$ is very close to SMLNUM or very close to BIGNUM, the matrix $A$ should be scaled.
See Also
Error Analysis
Matrix Storage Schemes

?poequb
Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix and reduce its condition number.

Syntax

```c
lapack_int LAPACKE_spoequb( int matrix_layout, lapack_int n, const float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```c
lapack_int LAPACKE_dpoequb( int matrix_layout, lapack_int n, const double* a, lapack_int lda, double* s, double* scond, double* amax );
```

```c
lapack_int LAPACKE_cpoequb( int matrix_layout, lapack_int n, const lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```c
lapack_int LAPACKE_zpoequb( int matrix_layout, lapack_int n, const lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive-definite matrix \( A \) and reduce its condition number (with respect to the two-norm).

These factors are chosen so that the scaled matrix \( B \) with elements \( B_{ij} = s_{i-1}A_{ij}s_{j-1} \) has diagonal elements equal to 1. \( s[i - 1] \) is a power of two nearest to, but not exceeding \( 1/sqrt(A_{ii}) \).

This choice of \( s \) puts the condition number of \( B \) within a factor \( n \) of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `n`
  The order of the matrix \( A \); \( n \geq 0 \).

- `a`
  Array: size max(1, lda*n).
  Contains the \( n \)-by-\( n \) symmetric or Hermitian positive definite matrix \( A \) whose scaling factors are to be computed. Only the diagonal elements of \( A \) are referenced.

- `lda`
  The leading dimension of \( a \); \( lda \geq max(1, m) \).

Output Parameters

- `s`
  Array, size \((n)\).
  If `info = 0`, the array \( s \) contains the scale factors for \( A \).
If info = 0, scond contains the ratio of the smallest $s[i]$ to the largest $s[i]$. If $scond \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by $s$.

$amax$

Absolute value of the largest element of the matrix $A$. If $amax$ is very close to SMLNUM or BIGNUM, the matrix should be scaled.

Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, parameter i had an illegal value.

If info = i, the $i$-th diagonal element of $A$ is nonpositive.

See Also

Error Analysis

Matrix Storage Schemes

?ppequ

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite matrix in packed storage and reduce its condition number.

Syntax

```c
lapack_int LAPACKE_sppequ( int matrix_layout, char uplo, lapack_int n, const float* ap, float* s, float* scond, float* amax );
lapack_int LAPACKE_dppequ( int matrix_layout, char uplo, lapack_int n, const double* ap, double* s, double* scond, double* amax );
lapack_int LAPACKE_cppequ( int matrix_layout, char uplo, lapack_int n, const lapack_complex_float* ap, float* s, float* scond, float* amax );
lapack_int LAPACKE_zppequ( int matrix_layout, char uplo, lapack_int n, const lapack_complex_double* ap, double* s, double* scond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite matrix $A$ in packed storage and reduce its condition number (with respect to the two-norm). The output array $s$ returns scale factors such that contains

$$s(i) = \frac{1}{\sqrt{a_{ii}}}$$

These factors are chosen so that the scaled matrix $B$ with elements $b_{ij}=s[i-1]*a_{ij}*s[j-1]$ has diagonal elements equal to 1.

This choice of $s$ puts the condition number of $B$ within a factor $n$ of the smallest possible condition number over all possible diagonal scalings.
Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo
Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of A is packed in the array ap:
If uplo = 'U', the array ap stores the upper triangular part of the matrix A.
If uplo = 'L', the array ap stores the lower triangular part of the matrix A.

n
The order of matrix A; n ≥ 0.

ap
Array, size at least max(1, n(n+1)/2). The array ap contains the upper or the lower triangular part of the matrix A (as specified by uplo) in packed storage (see Matrix Storage Schemes).

Output Parameters

s
Array, size (n).
If info = 0, the array s contains the scale factors for A.

scond
If info = 0, scond contains the ratio of the smallest s[i] to the largest s[i].

amax
Absolute value of the largest element of the matrix A.

Return Values

This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, the i-th diagonal element of A is nonpositive.

Application Notes

If scond ≥ 0.1 and amax is neither too large nor too small, it is not worth scaling by s.
If amax is very close to SMLNUM or very close to BIGNUM, the matrix A should be scaled.

See Also
Error Analysis
Matrix Storage Schemes

?pbequ
Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive-definite band matrix and reduce its condition number.

Syntax

lapack_int LAPACKE_spbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const float* ab, lapack_int ldab, float* s, float* scond, float* amax );
lapack_int LAPACKE_dpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const double* ab, lapack_int ldab, double* s, double* scond, double* amax );
lapack_int LAPACKE_cpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const lapack_complex_float* ab, lapack_int ldab, float* s, float* scond, float* amax );
lapack_int LAPACKE_zpbequ( int matrix_layout, char uplo, lapack_int n, lapack_int kd, const lapack_complex_double* ab, lapack_int ldab, double* s, double* scond, double* amax );

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric (Hermitian) positive definite band matrix \( A \) and reduce its condition number (with respect to the two-norm). The output array \( s \) returns scale factors such that contains

\[
S(i) = \frac{1}{\sqrt{a_{i,i}}}
\]

These factors are chosen so that the scaled matrix \( B \) with elements \( b_{ij} = s_{i-1} \cdot a_{ij} \cdot s_{j-1} \) has diagonal elements equal to 1. This choice of \( s \) puts the condition number of \( B \) within a factor \( n \) of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of \( A \) is stored in the array \( ab \):
  - If \( uplo = \text{'U'} \), the array \( ab \) stores the upper triangular part of the matrix \( A \).
  - If \( uplo = \text{'L'} \), the array \( ab \) stores the lower triangular part of the matrix \( A \).
- **n**: The order of matrix \( A \); \( n \geq 0 \).
- **kd**: The number of superdiagonals or subdiagonals in the matrix \( A \); \( kd \geq 0 \).
- **ab**: Array, size \( \max(1, ldab \cdot n) \).
  - The array \( ab \) contains either the upper or the lower triangular part of the matrix \( A \) (as specified by \( uplo \)) in band storage (see Matrix Storage Schemes).
- **ldab**: The leading dimension of the array \( ab \); \( ldab \geq kd + 1 \).
Output Parameters

$s$  
Array, size ($n$).  
If $info = 0$, the array $s$ contains the scale factors for $A$.

$scond$  
If $info = 0$, $scond$ contains the ratio of the smallest $s[i]$ to the largest $s[i]$.

$amax$  
Absolute value of the largest element of the matrix $A$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, the $i$-th diagonal element of $A$ is nonpositive.

Application Notes

If $scond < 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by $s$.

If $amax$ is very close to $SMLNUM$ or very close to $BIGNUM$, the matrix $A$ should be scaled.

See Also

Error Analysis
Matrix Storage Schemes

?syequb

Computes row and column scaling factors intended to equilibrate a symmetric indefinite matrix and reduce its condition number.

Syntax

lapack_int LAPACKE_ssyequb( int matrix_layout, char uplo, lapack_int $n$, const float* $a$, lapack_int lda, float* $s$, float* $scond$, float* $amax$);

lapack_int LAPACKE_dsyequb( int matrix_layout, char uplo, lapack_int $n$, const double* $a$, lapack_int lda, double* $s$, double* $scond$, double* $amax$);

lapack_int LAPACKE csyequb( int matrix_layout, char uplo, lapack_int $n$, const lapack_complex_float* $a$, lapack_int lda, float* $s$, float* $scond$, float* $amax$);

lapack_int LAPACKE_zsyequb( int matrix_layout, char uplo, lapack_int $n$, const lapack_complex_double* $a$, lapack_int lda, double* $s$, double* $scond$, double* $amax$);

Include Files

• mkl.h

Description

The routine computes row and column scalings intended to equilibrate a symmetric indefinite matrix $A$ and reduce its condition number (with respect to the two-norm).

The array $s$ contains the scale factors, $s[i-1] = 1/\sqrt{A(i,i)}$. These factors are chosen so that the scaled matrix $B$ with elements $b_{i,j} = s[i-1]*a_{i,j}*s[j-1]$ has ones on the diagonal.
This choice of $s$ puts the condition number of $B$ within a factor $n$ of the smallest possible condition number over all possible diagonal scalings.

**Input Parameters**

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  - Must be 'U' or 'L'.
  - Indicates whether the upper or lower triangular part of $A$ is stored:
    - If $uplo = 'U'$, the array $a$ stores the upper triangular part of the matrix $A$.
    - If $uplo = 'L'$, the array $a$ stores the lower triangular part of the matrix $A$.

- **n**
  - The order of the matrix $A$; $n \geq 0$.

- **a**
  - Array $a$: $\max(1, lda*n)$.
  - Contains the $n$-by-$n$ symmetric indefinite matrix $A$ whose scaling factors are to be computed. Only the diagonal elements of $A$ are referenced.

- **lda**
  - The leading dimension of $a$; $lda \geq \max(1, m)$.

**Output Parameters**

- **s**
  - Array, size ($n$).
  - If $info = 0$, the array $s$ contains the scale factors for $A$.

- **scond**
  - If $info = 0$, $scond$ contains the ratio of the smallest $s[i]$ to the largest $s[i]$. If $scond \geq 0.1$, and $amax$ is neither too large nor too small, it is not worth scaling by $s$.

- **amax**
  - Absolute value of the largest element of the matrix $A$. If $amax$ is very close to SMLNUM or BIGNUM, the matrix should be scaled.

**Return Values**

This function returns a value $info$.

- If $info = 0$, the execution is successful.
- If $info = -i$, parameter $i$ had an illegal value.
- If $info = i$, the $i$-th diagonal element of $A$ is nonpositive.

**See Also**

- Error Analysis
- Matrix Storage Schemes

?heequb

*Computes row and column scaling factors intended to equilibrate a Hermitian indefinite matrix and reduce its condition number.*
Syntax

```c
lapack_int LAPACKE_cheequb( int matrix_layout, char uplo, lapack_int n, const 
lapack_complex_float* a, lapack_int lda, float* s, float* scond, float* amax );
```

```c
lapack_int LAPACKE_zheequb( int matrix_layout, char uplo, lapack_int n, const 
lapack_complex_double* a, lapack_int lda, double* s, double* scond, double* amax );
```

Include Files

- mkl.h

Description

The routine computes row and column scalings intended to equilibrate a Hermitian indefinite matrix $A$ and reduce its condition number (with respect to the two-norm).

The array $s$ contains the scale factors, $s[i-1] = 1/\sqrt{a_{i,i}}$. These factors are chosen so that the scaled matrix $B$ with elements $b_{i,j} = s[i-1]a_{i,j}s[j-1]$ has ones on the diagonal.

This choice of $s$ puts the condition number of $B$ within a factor $n$ of the smallest possible condition number over all possible diagonal scalings.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of $A$ is stored:
  - If uplo = 'U', the array $a$ stores the upper triangular part of the matrix $A$.
  - If uplo = 'L', the array $a$ stores the lower triangular part of the matrix $A$.

- **n**
  The order of the matrix $A$; $n \geq 0$.

- **a**
  Array $a$: size max(1, lda*n).
  Contains the $n$-by-$n$ symmetric indefinite matrix $A$ whose scaling factors are to be computed. Only the diagonal elements of $A$ are referenced.

- **lda**
  The leading dimension of $a$; lda $\geq$ max(1, $m$).

Output Parameters

- **s**
  Array, size ($n$).
  If info = 0, the array $s$ contains the scale factors for $A$.

- **scond**
  If info = 0, scond contains the ratio of the smallest $s[i]$ to the largest $s[i]$. If scond $\geq$ 0.1, and amax is neither too large nor too small, it is not worth scaling by s.

- **amax**
  Absolute value of the largest element of the matrix $A$. If amax is very close to SMLNUM or BIGNUM, the matrix should be scaled.
Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, the i-th diagonal element of A is nonpositive.

See Also
Error Analysis
Matrix Storage Schemes

LAPACK Linear Equation Driver Routines
Table "Driver Routines for Solving Systems of Linear Equations" lists the LAPACK driver routines for solving systems of linear equations with real or complex matrices.

<table>
<thead>
<tr>
<th>Matrix type, storage scheme</th>
<th>Simple Driver</th>
<th>Expert Driver</th>
<th>Expert Driver using Extra-Precise Interactive Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>?gesv</td>
<td>?gesvx</td>
<td>?gesvxx</td>
</tr>
<tr>
<td>general band</td>
<td>?gbsv</td>
<td>?gbsvx</td>
<td>?gbsvxx</td>
</tr>
<tr>
<td>general tridiagonal</td>
<td>?gtsv</td>
<td>?gtsvx</td>
<td></td>
</tr>
<tr>
<td>diagonally dominant tridiagonal</td>
<td>?dtsvb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite</td>
<td>?posv</td>
<td>?posvx</td>
<td>?posvxx</td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite, storage</td>
<td>?ppsv</td>
<td>?ppsvx</td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite, band</td>
<td>?pbsv</td>
<td>?pbsvx</td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite, tridiagonal</td>
<td>?ptsv</td>
<td>?ptsvx</td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian indefinite</td>
<td>?sysv/?hesv</td>
<td>?sysvx/?hesvx</td>
<td>?sysvxx/?hesvxx</td>
</tr>
<tr>
<td>symmetric/Hermitian indefinite, packed storage</td>
<td>?sysv_rook/?sysv_rk/ ?hesv_rk</td>
<td>?sysv_aa/?hesv_aa</td>
<td></td>
</tr>
<tr>
<td>complex symmetric</td>
<td>?sysv</td>
<td>?sysvx</td>
<td>?sysv_rook</td>
</tr>
</tbody>
</table>
In this table ? stands for s (single precision real), d (double precision real), c (single precision complex), or z (double precision complex). In the description of ?gesv and ?posv routines, the ? sign stands for combined character codes ds and zc for the mixed precision subroutines.

**?gesv**

*Computes the solution to the system of linear equations with a square coefficient matrix A and multiple right-hand sides.*

**Syntax**

```c
lapack_int LAPACKE_sgesv (int matrix_layout, lapack_int n, lapack_int nrhs, float *a, lapack_int lda, lapack_int *ipiv, float *b, lapack_int ldb);

lapack_int LAPACKE_dgesv (int matrix_layout, lapack_int n, lapack_int nrhs, double *a, lapack_int lda, lapack_int *ipiv, double *b, lapack_int ldb);

lapack_int LAPACKE_cgesv (int matrix_layout, lapack_int n, lapack_int nrhs, lapack_complex_float *a, lapack_int lda, lapack_int *ipiv, lapack_complex_float *b, lapack_int ldb);

lapack_int LAPACKE_zgesv (int matrix_layout, lapack_int n, lapack_int nrhs, lapack_complex_double *a, lapack_int lda, lapack_int *ipiv, lapack_complex_double *b, lapack_int ldb);

lapack_int LAPACKE_dsgesv (int matrix_layout, lapack_int n, lapack_int nrhs, double *a, lapack_int lda, lapack_int *ipiv, double *b, lapack_int ldb, double *x, lapack_int ldx, lapack_int *iter);

lapack_int LAPACKE_zcgesv (int matrix_layout, lapack_int n, lapack_int nrhs, lapack_complex_double *a, lapack_int lda, lapack_int *ipiv, lapack_complex_double *b, lapack_int ldb, lapack_complex_double *x, lapack_int ldx, lapack_int *iter);
```

**Include Files**

- mkl.h

**Description**

The routine solves for X the system of linear equations $A^*X = B$, where $A$ is an $n$-by-$n$ matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions. The LU decomposition with partial pivoting and row interchanges is used to factor $A$ as $A = P*L*U$, where $P$ is a permutation matrix, $L$ is unit lower triangular, and $U$ is upper triangular. The factored form of $A$ is then used to solve the system of equations $A^*X = B$.

The dsgesv and zcgessv are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (dsgesv) or single complex precision (zcgesv) and use this factorization within an iterative refinement procedure to produce a solution with double precision (dsgesv) / double complex precision (zcgesv) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.
The iterative refinement is not going to be a winning strategy if the ratio single precision performance over double precision performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to ilaenv in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if

\[ \text{iter} > \text{itermax} \]

or for all the right-hand sides:

\[ \text{rnmr} < \sqrt{n} \times \text{xnrm} \times \text{anrm} \times \text{eps} \times \text{bwmax} \]

where

- \text{iter} is the number of the current iteration in the iterative refinement process
- \text{rnmr} is the infinity-norm of the residual
- \text{xnrm} is the infinity-norm of the solution
- \text{anrm} is the infinity-operator-norm of the matrix \( A \)
- \text{eps} is the machine epsilon returned by \text{dlamch} (‘Epsilon’).

The values \text{itermax} and \text{bwmax} are fixed to 30 and 1.0d+00 respectively.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **n**
  The number of linear equations, that is, the order of the matrix \( A \); \( n \geq 0 \).
- **nrhs**
  The number of right-hand sides, that is, the number of columns of the matrix \( B \); \( nrhs \geq 0 \).
- **a**
  The array \( a \) of size \( \max(1, lda \times n) \) contains the \( n \)-by-\( n \) coefficient matrix \( A \).
- **b**
  The array \( b \) of size \( \max(1, ldb \times nrhs) \) for column major layout and \( \max(1, ldb \times n) \) for row major layout contains the \( n \)-by-\( nrhs \) matrix of right hand side matrix \( B \).
- **lda**
  The leading dimension of the array \( a \); \( lda \geq \max(1, n) \).
- **ldb**
  The leading dimension of the array \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.
- **ldx**
  The leading dimension of the array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

**Output Parameters**

- **a**
  Overwritten by the factors \( L \) and \( U \) from the factorization of \( A = P*L*U \); the unit diagonal elements of \( L \) are not stored.

  If iterative refinement has been successfully used (\text{info} = 0 and \text{iter} \geq 0), then \( A \) is unchanged.
If double precision factorization has been used (\( \text{info} = 0 \) and \( \text{iter} < 0 \)), then the array \( A \) contains the factors \( L \) and \( U \) from the factorization \( A = P*L*U \); the unit diagonal elements of \( L \) are not stored.

\[ b \]
Overwritten by the solution matrix \( X \) for \( \text{dgesv} \), \( \text{sgevs} \), \( \text{zgesv} \), \( \text{zgesv} \). Unchanged for \( \text{dsgevs} \) and \( \text{zcgesv} \).

\[ \text{ipiv} \]
Array, size at least \( \max(1, n) \). The pivot indices that define the permutation matrix \( P \); row \( i \) of the matrix was interchanged with row \( \text{ipiv}[i-1] \). Corresponds to the single precision factorization (if \( \text{info} = 0 \) and \( \text{iter} \geq 0 \)) or the double precision factorization (if \( \text{info} = 0 \) and \( \text{iter} < 0 \)).

\[ x \]
Array, size \( \max(1, ldx*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout. If \( \text{info} = 0 \), contains the \( n \)-by-\( nrhs \) solution matrix \( X \).

\[ \text{iter} \]
If \( \text{iter} < 0 \): iterative refinement has failed. Iterative refinement has been performed

- If \( \text{iter} = -1 \): the routine fell back to full precision for implementation- or machine-specific reason
- If \( \text{iter} = -2 \): narrowing the precision induced an overflow, the routine fell back to full precision
- If \( \text{iter} = -3 \): failure of \( \text{sgetrf} \) for \( \text{dgesv} \), or \( \text{cgetrf} \) for \( \text{zcgesv} \)
- If \( \text{iter} = -31 \): stop the iterative refinement after the 30th iteration.

If \( \text{iter} > 0 \): iterative refinement has been successfully used. Returns the number of iterations.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i, U_{ii} \) (computed in double precision for mixed precision subroutines) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution could not be computed.

**See Also**

- \( \text{dlamch} \)
- \( \text{sgetrf} \)

**Matrix Storage Schemes**

\( \text{gesvx} \)

*Computes the solution to the system of linear equations with a square coefficient matrix \( A \) and multiple right-hand sides, and provides error bounds on the solution.*
Syntax

```c
lapack_int LAPACKE_dgesvx( int matrix_layout, char fact, char trans, lapack_int n,
    lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv,
    char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int ldx,
    float* rcond, float* ferr, float* berr, float* rpivot );

lapack_int LAPACKE_dgesvx( int matrix_layout, char fact, char trans, lapack_int n,
    lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
    lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c,
    lapack_int ldb, lapack_complex_float* x, lapack_int ldx,
    float* rcond, float* ferr, float* berr, float* rpivot );

lapack_int LAPACKE_zgesvx( int matrix_layout, char fact, char trans, lapack_int n,
    lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
    lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c,
    lapack_int ldb, lapack_complex_double* x, lapack_int ldx,
    double* rcond, double* ferr, double* berr, double* rpivot );
```

Include Files

- mkl.h

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations

$$AX = B,$$

where $A$ is an $n$-by-$n$ matrix, the columns of matrix $B$ are individual right-hand sides, and the
columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gesvx performs the following steps:

1. If `fact = 'E'`, real scaling factors $r$ and $c$ are computed to equilibrate the system:
   
   - $\text{trans} = 'N'$: $\text{diag}(r)^*A^{\text{trans}}\text{diag}(c)^{-1}X = \text{diag}(r)^*B$
   - $\text{trans} = 'T'$: $(\text{diag}(r)^*A^{\text{trans}}\text{diag}(c)^{-1})X = \text{diag}(r)^*B$
   - $\text{trans} = 'C'$: $(\text{diag}(r)^*A^{\text{trans}}\text{diag}(c)^{-1})X = \text{diag}(c)^*B$

   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if
   equilibration is used, $A$ is overwritten by $\text{diag}(r)^*A^{\text{trans}}\text{diag}(c)$ and $B$
   by $\text{diag}(r)^*B$ (if $\text{trans} = 'N'$) or $\text{diag}(c)^*B$ (if $\text{trans} = 'T'$ or 'C').

2. If `fact = 'N'` or 'E', the LU decomposition is used to factor the matrix $A$ (after equilibration if `fact`
   = 'E') as $A = P \cdot L \cdot U$, where $P$ is a permutation matrix, $L$ is a unit lower triangular matrix, and $U$
is upper triangular.

3. If some $U_{ii} = 0$, so that $U$ is exactly singular, then the routine returns with `info = i`. Otherwise, the
   factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the
   condition number is less than machine precision, `info = n + 1` is returned as a warning, but the
   routine still goes on to solve for $X$ and compute error bounds as described below.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and
   backward error estimates for it.
6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or 'C') so that it solves the original system before equilibration.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **fact**
  Must be 'F', 'N', or 'E'.
  Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.
  
  If $\text{fact} = 'F'$: on entry, $af$ and $ipiv$ contain the factored form of $A$. If $\text{equed}$ is not 'N', the matrix $A$ has been equilibrated with scaling factors given by $r$ and $c$. $a$, $af$, and $ipiv$ are not modified.
  
  If $\text{fact} = 'N'$, the matrix $A$ will be copied to $af$ and factored.
  
  If $\text{fact} = 'E'$, the matrix $A$ will be equilibrated if necessary, then copied to $af$ and factored.

- **trans**
  Must be 'N', 'T', or 'C'.
  Specifies the form of the system of equations:
  
  If $\text{trans} = 'N'$, the system has the form $A*X = B$ (No transpose).
  
  If $\text{trans} = 'T'$, the system has the form $A^T*X = B$ (Transpose).
  
  If $\text{trans} = 'C'$, the system has the form $A^H*X = B$ (Transpose for real flavors, conjugate transpose for complex flavors).

- **n**
  The number of linear equations; the order of the matrix $A$; $n \geq 0$.

- **nrhs**
  The number of right hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$.

- **a**
  The array $a$ of size $\text{max}(1, lda*n)$ contains the matrix $A$. If $\text{fact} = 'F'$ and $\text{equed}$ is not 'N', then $A$ must have been equilibrated by the scaling factors in $r$ and/or $c$.

- **af**
  The array $af$ of size $\text{max}(1, ldaf*n)$ is an input argument if $\text{fact} = 'F'$. It contains the factored form of the matrix $A$, that is, the factors $L$ and $U$ from the factorization $A = P*L*U$ as computed by ?getrf. If $\text{equed}$ is not 'N', then $af$ is the factored form of the equilibrated matrix $A$.

- **b**
  The array $b$ of size $\text{max}(1, ldb*nrhs)$ for column major layout and $\text{max}(1, ldb*n)$ for row major layout contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

- **lda**
  The leading dimension of $a$; $lda \geq \text{max}(1, n)$.

- **ldaf**
  The leading dimension of $af$; $ldaf \geq \text{max}(1, n)$.

- **ldb**
  The leading dimension of $b$; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
ipiv

Array, size at least \( \max(1, n) \). The array \( \text{ipiv} \) is an input argument if \( \text{fact} = 'F' \). It contains the pivot indices from the factorization \( A = PLU \) as computed by \texttt{?getrf}; row \( i \) of the matrix was interchanged with row \( \text{ipiv}[i-1] \).

equed

Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if \( \text{fact} = 'F' \). It specifies the form of equilibration that was done:

If \( \text{equed} = 'N' \), no equilibration was done (always true if \( \text{fact} = 'N' \)).

If \( \text{equed} = 'R' \), row equilibration was done, that is, \( A \) has been premultiplied by \( \text{diag}(r) \).

If \( \text{equed} = 'C' \), column equilibration was done, that is, \( A \) has been postmultiplied by \( \text{diag}(c) \).

If \( \text{equed} = 'B' \), both row and column equilibration was done, that is, \( A \) has been replaced by \( \text{diag}(r)^*A*\text{diag}(c) \).

\( r, c \)

Arrays: \( r \) (size \( n \)), \( c \) (size \( n \)). The array \( r \) contains the row scale factors for \( A \), and the array \( c \) contains the column scale factors for \( A \). These arrays are input arguments if \( \text{fact} = 'F' \) only; otherwise they are output arguments.

If \( \text{equed} = 'R' \) or 'B', \( A \) is multiplied on the left by \( \text{diag}(r) \); if \( \text{equed} = 'N' \) or 'C', \( r \) is not accessed.

If \( \text{fact} = 'F' \) and \( \text{equed} = 'R' \) or 'B', each element of \( r \) must be positive.

If \( \text{equed} = 'C' \) or 'B', \( A \) is multiplied on the right by \( \text{diag}(c) \); if \( \text{equed} = 'N' \) or 'R', \( c \) is not accessed.

If \( \text{fact} = 'F' \) and \( \text{equed} = 'C' \) or 'B', each element of \( c \) must be positive.

\( ldx \)

The leading dimension of the output array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq \text{nrhs} \) for row major layout.

Output Parameters

\( x \)

Array, size \( \max(1, ldx*\text{nrhs}) \) for column major layout and \( \max(1, ldx*n) \) for row major layout.

If \( \text{info} = 0 \) or \( \text{info} = n+1 \), the array \( x \) contains the solution matrix \( X \) to the original system of equations. Note that \( A \) and \( B \) are modified on exit if \( \text{equed} = 'N' \), and the solution to the equilibrated system is:

\[
\text{diag}(C)^{-1}*X, \text{ if } \text{trans} = 'N' \text{ and } \text{equed} = 'C' \text{ or 'B'};
\]

\[
\text{diag}(R)^{-1}*X, \text{ if } \text{trans} = 'T' \text{ or 'C'} \text{ and } \text{equed} = 'R' \text{ or 'B'}.\]

The second dimension of \( x \) must be at least \( \max(1, \text{nrhs}) \).

\( a \)

Array \( a \) is not modified on exit if \( \text{fact} = 'F' \) or 'N', or if \( \text{fact} = 'E' \) and \( \text{equed} = 'N' \). If \( \text{equed} = 'N' \), \( A \) is scaled on exit as follows:

\( \text{equed} = 'R': \ A = \text{diag}(R)^*A \)
equed = 'C': $A = A*\text{diag}(c)$
equed = 'B': $A = \text{diag}(R)*A*\text{diag}(c)$.

If $\text{fact} = 'N$' or 'E', then $af$ is an output argument and on exit returns the factors $L$ and $U$ from the factorization $A = PLU$ of the original matrix $A$ (if $\text{fact} = 'N$') or of the equilibrated matrix $A$ (if $\text{fact} = 'E$'). See the description of $a$ for the form of the equilibrated matrix.

*b*

Overwritten by $\text{diag}(r)*B$ if $\text{trans} = 'N$' and equed = 'R' or 'B';
overwritten by $\text{diag}(c)*B$ if $\text{trans} = 'T$' or 'C' and equed = 'C' or 'B';
not changed if equed = 'N'.

*r, c*

These arrays are output arguments if $\text{fact}$='F'. See the description of $r, c$ in Input Arguments section.

*rcond*

An estimate of the reciprocal condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. This condition is indicated by a return code of $\text{info} > 0$.

*ferr*

Array, size at least max(1, nrhs). Contains the estimated forward error bound for each solution vector $x_j$ (the $j$-th column of the solution matrix $X$). If $\text{xtrue}$ is the true solution corresponding to $x_j$, $\text{ferr}[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - \text{xtrue})$ divided by the magnitude of the largest element in $x_j$. The estimate is as reliable as the estimate for $\text{rcond}$, and is almost always a slight overestimate of the true error.

*berr*

Array, size at least max(1, nrhs). Contains the component-wise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

*ipiv*

If $\text{fact} = 'N$' or 'E', then $ipiv$ is an output argument and on exit contains the pivot indices from the factorization $A = P*L*U$ of the original matrix $A$ (if $\text{fact} = 'N$') or of the equilibrated matrix $A$ (if $\text{fact} = 'E$').

*equed*

If $\text{fact}='F$', then $\text{equed}$ is an output argument. It specifies the form of equilibration that was done (see the description of $\text{equed}$ in Input Arguments section).

*rpivot*

On exit, $\text{rpivot}$ contains the reciprocal pivot growth factor:

$$\frac{||A||}{||U||}$$

If $\text{rpivot}$ is much less than 1, then the stability of the $LU$ factorization of the (equilibrated) matrix $A$ could be poor. This also means that the solution $x$, condition estimator $\text{rcond}$, and forward
error bound \( \text{ferr} \) could be unreliable. If factorization fails with \( 0 < \text{info} \leq n \), then \( \text{rpivot} \) contains the reciprocal pivot growth factor for the leading \( \text{info} \) columns of \( A \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), parameter \( i \) had an illegal value.

If \( \text{info} = i \), and \( i \leq n \), then \( U(i, i) \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( \text{rcond} = 0 \) is returned.

If \( \text{info} = n + 1 \), then \( U \) is nonsingular, but \( \text{rcond} \) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \( \text{rcond} \) would suggest.

See Also

Matrix Storage Schemes

\(?\text{gesvxx}\)

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a square coefficient matrix \( A \) and multiple right-hand sides

Syntax

\[
\text{lapack_int LAPACKE_sgesvxx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );}
\]

\[
\text{lapack_int LAPACKE_dgesvxx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );}
\]

\[
\text{lapack_int LAPACKE_cgesvxx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );}
\]

\[
\text{lapack_int LAPACKE_zgesvxx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );}
\]
Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations
\( A \times X = B \), where \( A \) is an \( n \times n \) matrix, the columns of the matrix \( B \) are individual right-hand sides, and the
columns of \( X \) are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns
a solution with a small guaranteed error (\( O(\varepsilon) \), where \( \varepsilon \) is the working machine precision) unless the
matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also
calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the fact and
equed options. Solving with refinement and using a factorization from a previous call of the routine also
produces a solution with \( O(\varepsilon) \) errors or warnings but that may not be true for general user-provided
factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine \?gesvxx performs the following steps:

1. If \( \text{fact} = 'E' \), scaling factors \( r \) and \( c \) are computed to equilibrate the system:
   \[
   \begin{align*}
   \text{trans} = 'N': & \quad \text{diag}(r) \times A \times \text{diag}(c) \times \text{inv}(\text{diag}(c)) \times X = \text{diag}(r) \times B \\
   \text{trans} = 'T': & \quad (\text{diag}(r) \times A \times \text{diag}(c))^T \times \text{inv}(\text{diag}(r)) \times X = \text{diag}(c) \times B \\
   \text{trans} = 'C': & \quad (\text{diag}(r) \times A \times \text{diag}(c))^H \times \text{inv}(\text{diag}(r)) \times X = \text{inv}(\text{diag}(r)) \times B
   \end{align*}
   \]
   Whether or not the system will be equilibrated depends on the scaling of the matrix \( A \), but if
   equilibration is used, \( A \) is overwritten by \( \text{diag}(r) \times A \times \text{diag}(c) \) and \( B \) by \( \text{diag}(r) \times B \) (if \( \text{trans} = 'N' \) ) or
   \( \text{diag}(c) \times B \) (if \( \text{trans} = 'T' \) or 'C').
2. If \( \text{fact} = 'N' \) or 'E', the LU decomposition is used to factor the matrix \( A \) (after equilibration if \( \text{fact} = 'E' \) ) as \( A = P \times L \times U \), where \( P \) is a permutation matrix, \( L \) is a unit lower triangular matrix, and \( U \) is
   upper triangular.
3. If some \( U_{ii} = 0 \), so that \( U \) is exactly singular, then the routine returns with \( \text{info} = i \). Otherwise, the
   factored form of \( A \) is used to estimate the condition number of the matrix \( A \) (see the rcond parameter).
   If the reciprocal of the condition number is less than machine precision, the routine still goes on to
   solve for \( X \) and compute error bounds.
4. The system of equations is solved for \( X \) using the factored form of \( A \).
5. By default, unless is set to zero, the routine applies iterative refinement to improve the computed
   solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the
   working precision.
6. If equilibration was used, the matrix \( X \) is premultiplied by \( \text{diag}(c) \) (if \( \text{trans} = 'N' \) ) or \( \text{diag}(r) \) (if
   \( \text{trans} = 'T' \) or 'C') so that it solves the original system before equilibration.

Input Parameters

\textit{matrix\_layout} \hspace{1cm} Specifies whether matrix storage layout is row major
\hspace{1cm} \text{(LAPACK\_ROW\_MAJOR)} or column major \text{(LAPACK\_COL\_MAJOR)}.

\textit{fact} \hspace{1cm} Must be 'F', 'N', or 'E'.

\hspace{1cm} Specifies whether or not the factored form of the matrix \( A \) is supplied
\hspace{1cm} on entry, and if not, whether the matrix \( A \) should be equilibrated
\hspace{1cm} before it is factored.
If fact = 'F', on entry, af and ipiv contain the factored form of A. If equed is not 'N', the matrix A has been equilibrated with scaling factors given by r and c. Parameters a, af, and ipiv are not modified.

If fact = 'N', the matrix A will be copied to af and factored. If fact = 'E', the matrix A will be equilibrated, if necessary, copied to af and factored.

trans
Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:
If trans = 'N', the system has the form A*X = B (No transpose).
If trans = 'T', the system has the form A^T*X = B (Transpose).
If trans = 'C', the system has the form A^H*X = B (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).

n
The number of linear equations; the order of the matrix A; n ≥ 0.
nrhs
The number of right hand sides; the number of columns of the matrices B and X; nrhs ≥ 0.

a, af, b
Arrays: a(size max(lda*n)), af(size max(ldaf*n)), b(size max(1, ldb*nrhs) for column major layout and max(1, ldb*n) for row major layout).

The array a contains the matrix A. If fact = 'F' and equed is not 'N', then A must have been equilibrated by the scaling factors in r and/or c.

The array af is an input argument if fact = 'F'. It contains the factored form of the matrix A, that is, the factors L and U from the factorization A = P*L*U as computed by ?getrf. If equed is not 'N', then af is the factored form of the equilibrated matrix A.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

lda
The leading dimension of a; lda ≥ max(1, n).

ldaf
The leading dimension of af; ldaf ≥ max(1, n).

ipiv
Array, size at least max(1, n). The array ipiv is an input argument if fact = 'F'. It contains the pivot indices from the factorization A = P*L*U as computed by ?getrf; row i of the matrix was interchanged with row ipiv[i-1].

equed
Must be 'N', 'R', 'C', or 'B'.
equed is an input argument if fact = 'F'. It specifies the form of equilibration that was done:
If equed = 'N', no equilibration was done (always true if fact = 'N').
If $\text{equed} = 'R'$, row equilibration was done, that is, $A$ has been premultiplied by $\text{diag}(r)$.

If $\text{equed} = 'C'$, column equilibration was done, that is, $A$ has been postmultiplied by $\text{diag}(c)$.

If $\text{equed} = 'B'$, both row and column equilibration was done, that is, $A$ has been replaced by $\text{diag}(r) \cdot A \cdot \text{diag}(c)$.

Arrays: $r$ (size $n$), $c$ (size $n$). The array $r$ contains the row scale factors for $A$, and the array $c$ contains the column scale factors for $A$. These arrays are input arguments if $\text{fact} = 'F'$ only; otherwise they are output arguments.

If $\text{equed} = 'R'$ or $'B'$, $A$ is multiplied on the left by $\text{diag}(r)$; if $\text{equed} = 'N'$ or $'C'$, $r$ is not accessed.

If $\text{fact} = 'F'$ and $\text{equed} = 'R'$ or $'B'$, each element of $r$ must be positive.

If $\text{equed} = 'C'$ or $'B'$, $A$ is multiplied on the right by $\text{diag}(c)$; if $\text{equed} = 'N'$ or $'R'$, $c$ is not accessed.

If $\text{fact} = 'F'$ and $\text{equed} = 'C'$ or $'B'$, each element of $c$ must be positive.

Each element of $r$ or $c$ should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

`ldb` The leading dimension of the array $b$; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.

`ldx` The leading dimension of the output array $x$; $\text{ldx} \geq \max(1, n)$ for column major layout and $\text{ldx} \geq \text{nrhs}$ for row major layout.

`n_err_bnds` Number of error bounds to return for each right hand side and each type (normwise or componentwise). See $\text{err_bnds_norm}$ and $\text{err_bnds_comp}$ descriptions in *Output Arguments* section below.

`nparams` Specifies the number of parameters set in `params`. If $\leq 0$, the `params` array is never referenced and default values are used.

`params` Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to $\text{nparams}$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $\text{nparams} = 0$, which prevents the source code from accessing the `params` argument.

`params[0]`: Whether to perform iterative refinement or not. Default: 1.0

=0.0 No refinement is performed and no error bounds are computed.
Use the double-precision refinement algorithm, possibly with doubled-single computations if the compilation environment does not support double precision.

(Other values are reserved for future use.)

`params[1]`: Maximum number of residual computations allowed for refinement.

**Default**

10.0

**Aggressive**

Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]`: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

**Output Parameters**

*x*

Array, size `max(1, idx*nrhs)` for column major layout and `max(1, ldx*n)` for row major layout.

If `info = 0`, the array `x` contains the solution `n-by-nrhs` matrix `X` to the original system of equations. Note that `A` and `B` are modified on exit if `equed` is 'N', and the solution to the equilibrated system is:

- `inv(diag(c))*X`, if `trans = 'N'` and `equed = 'C'` or 'B'; or
- `inv(diag(r))*X`, if `trans = 'T'` or 'C' and `equed = 'R'` or 'B'.

*a*

Array `a` is not modified on exit if `fact = 'F'` or 'N', or if `fact = 'E'` and `equed = 'N'`.

If `equed` is 'N', `A` is scaled on exit as follows:

- `equed = 'R'`: `A = diag(r)*A`
- `equed = 'C'`: `A = A*diag(c)`
- `equed = 'B'`: `A = diag(r)*A*diag(c)`.

*af*

If `fact = 'N'` or 'E', then *af* is an output argument and on exit returns the factors `L` and `U` from the factorization `A = PLU` of the original matrix `A` (if `fact = 'N'`) or of the equilibrated matrix `A` (if `fact = 'E'`). See the description of `a` for the form of the equilibrated matrix.

*b*

Overwritten by `diag(r)*B` if `trans = 'N'` and `equed = 'R'` or 'B';
overwritten by `trans = 'T'` or 'C' and `equed = 'C'` or 'B';
not changed if `equed = 'N'`. 
These arrays are output arguments if `fact` ≠ `'F'`. Each element of these arrays is a power of the radix. See the description of `r, c` in `Input Arguments` section.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix `A` after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond = 0`, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

Contains the reciprocal pivot growth factor:

\[ \|A\|/\|C\| \]

If this is much less than 1, the stability of the `LU` factorization of the (equilibrated) matrix `A` could be poor. This also means that the solution `X`, estimated condition numbers, and error bounds could be unreliable. If factorization fails with `0 < info ≤ n`, this parameter contains the reciprocal pivot growth factor for the leading `info` columns of `A`. In `gesvx`, this quantity is returned in `rpivot`.

Array, size at least \( \max(1, nrhs) \). Contains the componentwise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of `A` or `B` that makes \( x_j \) an exact solution.

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the \( i \)-th solution vector

\[ \max_j \left| \frac{X_{true,j} - X_{j,i}}{\max_j |X_{true,j}|} \right| \]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

- **err=1**
  "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors.

- **err=2**
  "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold `sqrt(n)*slamch(ε)` for single precision flavors and `sqrt(n)*dlamch(ε)` for double precision flavors. This error bound should only be trusted if the previous boolean is true.

- **err=3**
  Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold
Let \( z = s \cdot a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in \( \text{err}_bnds\_comp[ (\text{err}-1) \cdot \text{nrhs} + i - 1 ] \).

Array of size \( \text{nrhs} \cdot \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \( i \)-th solution vector:

\[
\max_j \frac{|X^{true}_{ji} - X_{ji}|}{|X_{ji}|}
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (\( \text{params}[2] = 0.0 \)), then \( \text{err}_bnds\_comp \) is not accessed.

\( \text{err}=1 \)

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors.

\( \text{err}=2 \)

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

\( \text{err}=3 \)

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is...
"guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

\[ ||Z||_\infty \cdot ||Z^{-1}||_\infty \]

Let \( z = s \cdot (a \cdot \text{diag}(x)) \), where \( x \) is the solution for the current right-hand side and \( s \) scales each row of \( a \cdot \text{diag}(x) \) by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in \( \text{err\_bnds\_comp}\left[(\text{err}-1)\times\text{nrhs} + i - 1\right] \).

- **ipiv**
  - If \( \text{fact} = 'N' \) or \( 'E' \), then \( \text{ipiv} \) is an output argument and on exit contains the pivot indices from the factorization \( A = P \cdot L \cdot U \) of the original matrix \( A \) (if \( \text{fact} = 'N' \)) or of the equilibrated matrix \( A \) (if \( \text{fact} = 'E' \)).

- **equed**
  - If \( \text{fact} \neq 'F' \), then \( \text{equed} \) is an output argument. It specifies the form of equilibration that was done (see the description of \( \text{equed} \) in Input Arguments section).

- **params**
  - If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.
- If \( \text{info} = -i \), parameter \( i \) had an illegal value.
- If \( 0 < \text{info} < \text{nrhs} \) \( \text{info} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( \text{rcond} = 0 \) is returned.
- If \( \text{info} = n + j \): The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest \( j \) such that for column major layout \( \text{err\_bnds\_norm}(j - 1) = 0.0 \) or \( \text{err\_bnds\_comp}(j - 1) = 0.0 \); or for row major layout \( \text{err\_bnds\_norm}[(j - 1) \times n_{\text{err\_bnds}}] = 0.0 \) or \( \text{err\_bnds\_comp}[(j - 1) \times n_{\text{err\_bnds}}] = 0.0 \)). See the definition of \( \text{err\_bnds\_norm} \) and \( \text{err\_bnds\_comp} \) for \( \text{err} = 1 \). To get information about all of the right-hand sides, check \( \text{err\_bnds\_norm} \) or \( \text{err\_bnds\_comp} \).

**See Also**

Matrix Storage Schemes

**?gbsv**

*Computes the solution to the system of linear equations with a band coefficient matrix A and multiple right-hand sides.*

**Syntax**

```c
lapack_int LAPACKE_sgbsv (int matrix_layout , lapack_int n , lapack_int kl , lapack_int ku , lapack_int nrhs , float * ab , lapack_int ldab , lapack_int * ipiv , float * b , lapack_int ldb );
```
The routine solves for $X$ the real or complex system of linear equations $A*X = B$, where $A$ is an $n$-by-$n$ band matrix with $kl$ subdiagonals and $ku$ superdiagonals, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

The $LU$ decomposition with partial pivoting and row interchanges is used to factor $A$ as $A = L*U$, where $L$ is a product of permutation and unit lower triangular matrices with $kl$ subdiagonals, and $U$ is upper triangular with $kl+ku$ superdiagonals. The factored form of $A$ is then used to solve the system of equations $A*X = B$.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **$n$**
  The order of $A$. The number of rows in $B$; $n \geq 0$.

- **$kl$**
  The number of subdiagonals within the band of $A$; $kl \geq 0$.

- **$ku$**
  The number of superdiagonals within the band of $A$; $ku \geq 0$.

- **$nrhs$**
  The number of right-hand sides. The number of columns in $B$; $nrhs \geq 0$.

- **$ab$, $b$**
  Arrays: $ab$ (size max(1, $ldab*n$)), $b$ (size max(1, $ldb*nrhs$)) for column major layout and max(1, $ldb*n$) for row major layout.

  The array $ab$ contains the matrix $A$ in band storage (see Matrix Storage Schemes).

  The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

- **$ldab$**
  The leading dimension of the array $ab$. ($ldab \geq 2kl + ku +1$)

- **$ldb$**
  The leading dimension of $b$; $ldb \geq max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

**Output Parameters**

- **$ab$**
  Overwritten by $L$ and $U$. $U$ is stored as an upper triangular band matrix with $kl + ku$ superdiagonals and $L$ is stored as a lower triangular band matrix with $kl$ subdiagonals. See Matrix Storage Schemes.
Overwritten by the solution matrix $X$.

Array, size at least \( \max(1, n) \). The pivot indices: row $i$ was interchanged with row $ipiv[i-1]$.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, $U_{ii}$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution could not be computed.

**See Also**

Matrix Storage Schemes

**?gbsvx**

Computes the solution to the real or complex system of linear equations with a band coefficient matrix $A$ and multiple right-hand sides, and provides error bounds on the solution.

**Syntax**

```c
lapack_int LAPACKE_sgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
                        lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
                        lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
                        lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr, float* rpivot );

lapack_int LAPACKE_dgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
                        lapack_int kl, lapack_int ku, lapack_int nrhs, double* ab, lapack_int ldab, double* afb,
                        lapack_int ldafb, lapack_int* ipiv, char* equed, double* r, double* c, double* b,
                        lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );

lapack_int LAPACKE_cgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
                        lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int ldab,
                        lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
                        float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
                        lapack_int ldx, float* rcond, float* ferr, float* berr, float* rpivot );

lapack_int LAPACKE_zgbsvx( int matrix_layout, char fact, char trans, lapack_int n,
                        lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int ldab,
                        lapack_complex_double* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
                        double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
                        lapack_int ldx, double* rcond, double* ferr, double* berr, double* rpivot );
```

**Include Files**

- mkl.h
Description

The routine uses the $LU$ factorization to compute the solution to a real or complex system of linear equations $A*X = B$, $A^TX = B$, or $A^H*X = B$, where $A$ is a band matrix of order $n$ with $kl$ subdiagonals and $ku$ superdiagonals, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `?gbsvx` performs the following steps:

1. If `fact = 'E'`, real scaling factors $r$ and $c$ are computed to equilibrate the system:
   
   $\text{trans} = 'N'$: $\text{diag}(r)*A\text{diag}(c) *\text{inv}(\text{diag}(c))*X = \text{diag}(r)*B$
   
   $\text{trans} = 'T'$: $(\text{diag}(r)*A\text{diag}(c))^T *\text{inv}(\text{diag}(r))*X = \text{diag}(c)*B$
   
   $\text{trans} = 'C'$: $(\text{diag}(r)*A\text{diag}(c))^H *\text{inv}(\text{diag}(r))*X = \text{diag}(c)*B$

   Whether the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(r)*A\text{diag}(c)$ and $B$ by $\text{diag}(r)*B$ (if $\text{trans} = 'N'$) or $\text{diag}(c)*B$ (if $\text{trans} = 'T'$ or 'C').

2. If `fact = 'N'` or 'E', the $LU$ decomposition is used to factor the matrix $A$ (after equilibration if `fact = 'E'`) as $A = L*U$, where $L$ is a product of permutation and unit lower triangular matrices with $kl$ subdiagonals, and $U$ is upper triangular with $kl+ku$ superdiagonals.

3. If some $U_{i,j} = 0$, so that $U$ is exactly singular, then the routine returns with $\text{info} = i$. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, $\text{info} = n + 1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(c)$ (if $\text{trans} = 'N'$) or $\text{diag}(r)$ (if $\text{trans} = 'T'$ or 'C') so that it solves the original system before equilibration.

Input Parameters

`matrix_layout`  
Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

`fact`  
Must be 'F', 'N', or 'E'.  

Specifies whether the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.  

If `fact = 'F'`: on entry, $ab$ and $ipiv$ contain the factored form of $A$.  

If `equed` is not 'N', the matrix $A$ is equilibrated with scaling factors given by $r$ and $c$.  

$ab$, $afb$, and $ipiv$ are not modified.  

If `fact = 'N'`, the matrix $A$ will be copied to $ab$ and factored.  

If `fact = 'E'`, the matrix $A$ will be equilibrated if necessary, then copied to $ab$ and factored.

`trans`  
Must be 'N', 'T', or 'C'.  

Specifies the form of the system of equations:  

If `trans = 'N'`, the system has the form $A*X = B$ (No transpose).
If \( \text{trans} = 'T' \), the system has the form \( A^T X = B \) (Transpose).

If \( \text{trans} = 'C' \), the system has the form \( A^\dagger X = B \) (Transpose for real flavors, conjugate transpose for complex flavors).

\( n \)

The number of linear equations, the order of the matrix \( A \); \( n \geq 0 \).

\( kl \)

The number of subdiagonals within the band of \( A \); \( kl \geq 0 \).

\( ku \)

The number of superdiagonals within the band of \( A \); \( ku \geq 0 \).

\( nrhs \)

The number of right hand sides, the number of columns of the matrices \( B \) and \( X \); \( nrhs \geq 0 \).

\( ab, afb, b \)

Arrays: \( ab \) (max(\( ldab \)*\( n \))), \( afb \) (max(\( ldafb \)*\( n \))), \( b \) (max(1, \( ldb \)*\( nrhs \)) for column major layout and max(1, \( ldb \)*\( n \)) for row major layout).

The array \( ab \) contains the matrix \( A \) in band storage (see Matrix Storage Schemes). If \( \text{fact} = 'F' \) and \( \text{equed} \) is not 'N', then \( A \) must have been equilibrated by the scaling factors in \( r \) and/or \( c \).

The array \( afb \) is an input argument if \( \text{fact} = 'F' \). It contains the factored form of the matrix \( A \), that is, the factors \( L \) and \( U \) from the factorization \( A = P*L*U \) as computed by \(?gbtrf\). \( U \) is stored as an upper triangular band matrix with \( kl + ku \) superdiagonals. \( L \) is stored as lower triangular band matrix with \( kl \) subdiagonals. If \( \text{equed} \) is not 'N', then \( afb \) is the factored form of the equilibrated matrix \( A \).

The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

\( ldab \)

The leading dimension of \( ab \); \( ldab \geq kl + ku + 1 \).

\( ldafb \)

The leading dimension of \( afb \); \( ldafb \geq 2*kl + ku + 1 \).

\( ldb \)

The leading dimension of \( b \); \( ldb \geq \max(1, \ n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

\( ipiv \)

Array, size at least \( \max(1, \ n) \). The array \( ipiv \) is an input argument if \( \text{fact} = 'F' \). It contains the pivot indices from the factorization \( A = P*L*U \) as computed by \(?gbtrf\); row \( i \) of the matrix was interchanged with row \( ipiv[i-1] \).

\( equed \)

Must be 'N', 'R', 'C', or 'B'.

\( equed \) is an input argument if \( \text{fact} = 'F' \). It specifies the form of equilibration that was done:

If \( \text{equed} = 'N' \), no equilibration was done (always true if \( \text{fact} = 'N' \)).

If \( \text{equed} = 'R' \), row equilibration was done, that is, \( A \) has been premultiplied by \( \text{diag}(r) \).

If \( \text{equed} = 'C' \), column equilibration was done, that is, \( A \) has been postmultiplied by \( \text{diag}(c) \).

If \( \text{equed} = 'B' \), both row and column equilibration was done, that is, \( A \) has been replaced by \( \text{diag}(r)*A*\text{diag}(c) \).
Arrays: \( r \) (size \( n \)), \( c \) (size \( n \)).

The array \( r \) contains the row scale factors for \( A \), and the array \( c \) contains the column scale factors for \( A \). These arrays are input arguments if \( \text{fact} = 'F' \) only; otherwise they are output arguments.

If \( \text{equed} = 'R' \) or 'B', \( A \) is multiplied on the left by \( \text{diag}(r) \); if \( \text{equed} = 'N' \) or 'C', \( r \) is not accessed.

If \( \text{fact} = 'F' \) and \( \text{equed} = 'R' \) or 'B', each element of \( r \) must be positive.

If \( \text{equed} = 'C' \) or 'B', \( A \) is multiplied on the right by \( \text{diag}(c) \); if \( \text{equed} = 'N' \) or 'R', \( c \) is not accessed.

If \( \text{fact} = 'F' \) and \( \text{equed} = 'C' \) or 'B', each element of \( c \) must be positive.

\( ldx \)

The leading dimension of the output array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq \text{nrhs} \) for row major layout.

Output Parameters

\( x \)

Array, size \( \max(1, ldx*\text{nrhs}) \) for column major layout and \( \max(1, ldx*n) \) for row major layout.

If \( \text{info} = 0 \) or \( \text{info} = n+1 \), the array \( x \) contains the solution matrix \( X \) to the original system of equations. Note that \( A \) and \( B \) are modified on exit if \( \text{equed} \neq 'N' \), and the solution to the equilibrated system is:

\[
\text{inv(\text{diag}(c))}*X, \text{if trans} = 'N' \text{ and } \text{equed} = 'C' \text{ or 'B'}; \\
\text{inv(\text{diag}(r))}*X, \text{if trans} = 'T' \text{ or 'C' and equed} = 'R' \text{ or 'B'.}
\]

\( ab \)

Array \( ab \) is not modified on exit if \( \text{fact} = 'F' \) or 'N', or if \( \text{fact} = 'E' \) and \( \text{equed} = 'N' \).

If \( \text{equed} \neq 'N' \), \( A \) is scaled on exit as follows:

\[
\text{equed} = 'R': A = \text{diag}(r) * A \\
\text{equed} = 'C': A = A * \text{diag}(c) \\
\text{equed} = 'B': A = \text{diag}(r) * A * \text{diag}(c).
\]

\( afb \)

If \( \text{fact} = 'N' \) or 'E', then \( afb \) is an output argument and on exit returns details of the \( LU \) factorization of the original matrix \( A \) (if \( \text{fact} = 'N' \)) or of the equilibrated matrix \( A \) (if \( \text{fact} = 'E' \)). See the description of \( ab \) for the form of the equilibrated matrix.

\( b \)

Overwritten by \( \text{diag}(r)*b \) if \( \text{trans} = 'N' \) and \( \text{equed} = 'R' \) or 'B';

overwritten by \( \text{diag}(c)*b \) if \( \text{trans} = 'T' \) or 'C' and \( \text{equed} = 'C' \) or 'B';

not changed if \( \text{equed} = 'N' \).

\( r, c \)

These arrays are output arguments if \( \text{fact} \neq 'F' \). See the description of \( r, c \) in Input Arguments section.

\( rcond \)

An estimate of the reciprocal condition number of the matrix \( A \) after equilibration (if done).
If \( rcond \) is less than the machine precision (in particular, if \( rcond = 0 \)), the matrix is singular to working precision. This condition is indicated by a return code of \( info > 0 \).

\textbf{ferr}

Array, size at least \( \max(1, nrhs) \). Contains the estimated forward error bound for each solution vector \( x_j \) (the \( j \)-th column of the solution matrix \( X \)). If \( xtrue \) is the true solution corresponding to \( x_j \), \( ferr[j-1] \) is an estimated upper bound for the magnitude of the largest element in \( (x_j - xtrue) \) divided by the magnitude of the largest element in \( x_j \). The estimate is as reliable as the estimate for \( rcond \), and is almost always a slight overestimate of the true error.

\textbf{berr}

Array, size at least \( \max(1, nrhs) \). Contains the component-wise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of \( A \) or \( B \) that makes \( x_j \) an exact solution.

\textbf{ipiv}

If \( fact = 'N' \) or \( 'E' \), then \( ipiv \) is an output argument and on exit contains the pivot indices from the factorization \( A = L*U \) of the original matrix \( A \) (if \( fact = 'N' \)) or of the equilibrated matrix \( A \) (if \( fact = 'E' \)).

\textbf{equed}

If \( fact \\# 'F' \), then \( equed \) is an output argument. It specifies the form of equilibration that was done (see the description of \( equed \) in Input Arguments section).

\textbf{rpivot}

On exit, \( rpivot \) contains the reciprocal pivot growth factor:

\[
\frac{\|A\|}{\|U\|}
\]

If \( rpivot \) is much less than 1, then the stability of the \( LU \) factorization of the (equilibrated) matrix \( A \) could be poor. This also means that the solution \( x \), condition estimator \( rcond \), and forward error bound \( ferr \) could be unreliable. If factorization fails with \( 0 < info \leq n \), then \( rpivot \) contains the reciprocal pivot growth factor for the leading \( info \) columns of \( A \).

\textbf{Return Values}

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), and \( i \leq n \), then \( U_{i,i} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( rcond = 0 \) is returned. If \( info = i \), and \( i = n+1 \), then \( U \) is nonsingular, but \( rcond \) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \( rcond \) would suggest.

\textbf{See Also}

Matrix Storage Schemes
?gbsvxx

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a banded coefficient matrix $A$ and multiple right-hand sides.

**Syntax**

```c
#include <mkl.h>

// Real Banded Matrix
lapack_int LAPACKE_dgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
                          lapack_int kl, lapack_int ku, lapack_int nrhs, float* ab, lapack_int ldab, float* afb,
                          lapack_int ldafb, lapack_int* ipiv, char* equed, float* r, float* c, float* b,
                          lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr,
                          lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams,
                          const float* params );

// Complex Banded Matrix
lapack_int LAPACKE_cgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
                           lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_float* ab, lapack_int ldab,
                           lapack_complex_float* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
                           float* r, float* c, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
                           lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds,
                           float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );

// Complex Hermitian Banded Matrix
lapack_int LAPACKE_zgbsvxx( int matrix_layout, char fact, char trans, lapack_int n,
                           lapack_int kl, lapack_int ku, lapack_int nrhs, lapack_complex_double* ab, lapack_int ldab,
                           lapack_complex_double* afb, lapack_int ldafb, lapack_int* ipiv, char* equed,
                           double* r, double* c, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x,
                           lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds,
                           double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

**Include Files**

- mkl.h

**Description**

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A^T X = B$, $A^H X = B$, or $A^H X = B$, where $A$ is an $n$-by-$n$ banded matrix, the columns of the matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\varepsilon)$), where $\varepsilon$ is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the `fact` and `equed` options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\varepsilon)$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.
The routine \( ?gbsvxx \) performs the following steps:

1. If \( \text{fact} = 'E' \), scaling factors \( r \) and \( c \) are computed to equilibrate the system:
   \[
   \begin{align*}
   \text{trans} = 'N': & \quad \text{diag}(r)A\text{diag}(c)\text{inv}(\text{diag}(c))X = \text{diag}(r)B \\
   \text{trans} = 'T': & \quad (\text{diag}(r)A\text{diag}(c))^2\text{inv}(\text{diag}(r))X = \text{diag}(c)B \\
   \text{trans} = 'C': & \quad (\text{diag}(r)A\text{diag}(c))^H\text{inv}(\text{diag}(r))X = \text{diag}(c)B
   \end{align*}
   \]
   Whether or not the system will be equilibrated depends on the scaling of the matrix \( A \), but if equilibration is used, \( A \) is overwritten by \( \text{diag}(r)A\text{diag}(c) \) and \( B \) by \( \text{diag}(r)B \) (if \( \text{trans} = 'N' \)) or \( \text{diag}(c)B \) (if \( \text{trans} = 'T' \) or \( 'C' \)).

2. If \( \text{fact} = 'N' \) or \( 'E' \), the \( LU \) decomposition is used to factor the matrix \( A \) (after equilibration if \( \text{fact} = 'E' \)) as \( A = PLU \), where \( P \) is a permutation matrix, \( L \) is a unit lower triangular matrix, and \( U \) is upper triangular.

3. If some \( U_{i,i} = 0 \), so that \( U \) is exactly singular, then the routine returns with \( \text{info} = i \). Otherwise, the factored form of \( A \) is used to estimate the condition number of the matrix \( A \) (see the \( \text{rcond} \) parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for \( X \) and compute error bounds.

4. The system of equations is solved for \( X \) using the factored form of \( A \).

5. By default, unless \( \text{params}[0] \) is set to zero, the routine applies iterative refinement to improve the computed solution matrix and calculate error bounds. Refinement calculates the residual to at least twice the working precision.

6. If equilibration was used, the matrix \( X \) is premultiplied by \( \text{diag}(c) \) (if \( \text{trans} = 'N' \)) or \( \text{diag}(r) \) (if \( \text{trans} = 'T' \) or \( 'C' \)) so that it solves the original system before equilibration.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether two-dimensional array storage is row major (\text{LAPACK\_ROW\_MAJOR}) or column major (\text{LAPACK\_COL\_MAJOR}).

- **fact**
  
  Must be \( 'F' \), \( 'N' \), or \( 'E' \).

  Specifies whether or not the factored form of the matrix \( A \) is supplied on entry, and if not, whether the matrix \( A \) should be equilibrated before it is factored.

  If \( \text{fact} = 'F' \), on entry, \( \text{afb} \) and \( \text{ipiv} \) contain the factored form of \( A \). If \( \text{equed} \) is not \( 'N' \), the matrix \( A \) has been equilibrated with scaling factors given by \( r \) and \( c \). Parameters \( \text{ab} \), \( \text{afb} \), and \( \text{ipiv} \) are not modified.

  If \( \text{fact} = 'N' \), the matrix \( A \) will be copied to \( \text{afb} \) and factored.

  If \( \text{fact} = 'E' \), the matrix \( A \) will be equilibrated, if necessary, copied to \( \text{afb} \) and factored.

- **trans**
  
  Must be \( 'N' \), \( 'T' \), or \( 'C' \).

  Specifies the form of the system of equations:

  If \( \text{trans} = 'N' \), the system has the form \( AX = B \) (No transpose).

  If \( \text{trans} = 'T' \), the system has the form \( A^T X = B \) (Transpose).

  If \( \text{trans} = 'C' \), the system has the form \( A^H X = B \) (Conjugate Transpose = Transpose for real flavors, Conjugate Transpose for complex flavors).

- **n**
  
  The number of linear equations; the order of the matrix \( A \); \( n \geq 0 \).
The number of subdiagonals within the band of $A$; $kl \geq 0$.

The number of superdiagonals within the band of $A$; $ku \geq 0$.

The number of right-hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$.

Arrays: $ab$ (max($ldab*ln$)), $afb$ (max($ldafb*ln$)), $b$ (max(1, $ldb*nrhs$) for column major layout and max(1, $ldb*n$) for row major layout).

The array $ab$ contains the matrix $A$ in band storage.

If $fact = 'F'$ and $equed$ is not 'N', then $AB$ must have been equilibrated by the scaling factors in $r$ and/or $c$.

The array $afb$ is an input argument if $fact = 'F'$. It contains the factored form of the banded matrix $A$, that is, the factors $L$ and $U$ from the factorization $A = P*L*U$ as computed by $?gbtrf$. $U$ is stored as an upper triangular banded matrix with $kl + ku$ superdiagonals. $L$ is stored as lower triangular band matrix with $kl$ subdiagonals. If $equed$ is not 'N', then $afb$ is the factored form of the equilibrated matrix $A$.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

The leading dimension of the array $ab$; $ldab \geq kl + ku + 1$.

The leading dimension of the array $afb$; $ldafb \geq 2*kl + ku + 1$.

Array, size at least max(1, $n$). The array $ipiv$ is an input argument if $fact = 'F'$. It contains the pivot indices from the factorization $A = P*L*U$ as computed by $?gbtrf$; row $i$ of the matrix was interchanged with row $ipiv[i-1]$.

Must be 'N', 'R', 'C', or 'B'.

equed is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:

If $equed$ = 'N', no equilibration was done (always true if $fact$ = 'N').

If $equed$ = 'R', row equilibration was done, that is, $A$ has been premultiplied by $diag(r)$.

If $equed$ = 'C', column equilibration was done, that is, $A$ has been postmultiplied by $diag(c)$.

If $equed$ = 'B', both row and column equilibration was done, that is, $A$ has been replaced by $diag(r)*A*diag(c)$.

Arrays: $r$ (size $n$), $c$ (size $n$). The array $r$ contains the row scale factors for $A$, and the array $c$ contains the column scale factors for $A$. These arrays are input arguments if $fact = 'F'$ only; otherwise they are output arguments.

If $equed$ = 'R' or 'B', $A$ is multiplied on the left by $diag(r)$; if $equed$ = 'N' or 'C', $r$ is not accessed.
If \( fact = 'F' \) and \( equed = 'R' \) or \( 'B' \), each element of \( r \) must be positive.

If \( equed = 'C' \) or \( 'B' \), \( A \) is multiplied on the right by \( \text{diag}(c) \); if \( equed = 'N' \) or \( 'R' \), \( c \) is not accessed.

If \( fact = 'F' \) and \( equed = 'C' \) or \( 'B' \), each element of \( c \) must be positive.

Each element of \( r \) or \( c \) should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

\( \text{ldb} \)

The leading dimension of the array \( b \); \( \text{ldb} \geq \max(1, n) \) for column major layout and \( \text{ldb} \geq \text{nrhs} \) for row major layout.

\( \text{ldx} \)

The leading dimension of the output array \( x \); \( \text{ldx} \geq \max(1, n) \) for column major layout and \( \text{ldx} \geq \text{nrhs} \) for row major layout.

\( n_{\text{err\_bnds}} \)

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See \( \text{err\_bnds\_norm} \) and \( \text{err\_bnds\_comp} \) descriptions in \textit{Output Arguments} section below.

\( n_{\text{params}} \)

Specifies the number of parameters set in \( \text{params} \). If \( \leq 0 \), the \( \text{params} \) array is never referenced and default values are used.

\( \text{params} \)

Array, size \( \max(1, n_{\text{params}}) \). Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to \( n_{\text{params}} \) are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass \( n_{\text{params}} = 0 \), which prevents the source code from accessing the \( \text{params} \) argument.

\( \text{params}[0] \) : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

\( =0.0 \)

No refinement is performed and no error bounds are computed.

\( =1.0 \)

Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

\( \text{params}[1] \) : Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive

Set to 100.0 to permit convergence using approximate factorizations or factorizations other than \( LU \). If the factorization uses a technique other than Gaussian elimination,
the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.

`params[2]` : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

**Output Parameters**

`x`  
Array, size max(1, `ldx*nrhs`) for column major layout and max(1, `ldx*n`) for row major layout.

If `info = 0`, the array `x` contains the solution `n`-by-`nrhs` matrix `X` to the original system of equations. Note that `A` and `B` are modified on exit if `equed≠'N'`, and the solution to the equilibrated system is:

\[
\text{inv}(\text{diag}(c)) \times X, \text{ if } \text{trans} = 'N' \text{ and equed = 'C' or 'B'}; \text{ or}
\]

\[
\text{inv}(\text{diag}(r)) \times X, \text{ if } \text{trans} = 'T' \text{ or 'C'} \text{ and equed = 'R' or 'B'}.
\]

`ab`  
Array `ab` is not modified on exit if `fact = 'F'` or 'N', or if `fact = 'E'` and `equed = 'N'`.

If `equed≠'N'`, `A` is scaled on exit as follows:

- `equed = 'R'`: \( A = \text{diag}(r) \times A \)
- `equed = 'C'`: \( A = A \times \text{diag}(c) \)
- `equed = 'B'`: \( A = \text{diag}(r) \times A \times \text{diag}(c) \).

`afb`  
If `fact = 'N'` or 'E', then `afb` is an output argument and on exit returns the factors `L` and `U` from the factorization \( A = PLU \) of the original matrix `A` (if `fact = 'N'`) or of the equilibrated matrix `A` (if `fact = 'E'`).

`b`  
Overwritten by `\text{diag}(r) \times B` if `trans = 'N'` and `equed = 'R'` or 'B';

overwritten by `\text{trans} = 'T'` or 'C' and `equed = 'C' or 'B';`

not changed if `equed = 'N'`.

`r, c`  
These arrays are output arguments if `fact≠'F'`. Each element of these arrays is a power of the radix. See the description of `r, c` in Input Arguments section.

`rcond`  
Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix `A` after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond = 0`, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

`rpvgrw`  
Contains the reciprocal pivot growth factor:

\[ \frac{\|A\|}{\|L\|\|U\|} \]

If this is much less than 1, the stability of the `LU` factorization of the (equilibrated) matrix `A` could be poor. This also means that the solution `X`, estimated condition numbers, and error bounds could be unreliable. If
factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading info columns of A. In gbsvx, this quantity is returned in rpivot.

*berr*

Array, size at least $\max(1, \text{nrhs})$. Contains the componentwise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

*err_bnds_norm*

Array of size $\text{nrhs} \times n\_err\_bnds$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the $i$-th solution vector

$$\frac{\max_j |X_{true,j} - X_{j}|}{\max_j |X_{j}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

*err=1*  
"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors.

*err=2*  
"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

*err=3*  
Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\| \cdot \|Z^{-1}\|_\infty$$

Let $z=s\times a$, where $s$ scales each row by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error err is stored in err_bnds_norm[(err-1)*nrhs + i - 1].
**err_bnds_comp**

Array of size $nrhs \times n_{\text{err\_bnds}}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the $i$-th solution vector:

$$\max_j \frac{|X_{true}_{ij} - X_{i,j}|}{|X_{i,j}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($params[2] = 0.0$), then `err_bnds_comp` is not accessed.

**err=1**

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} \times \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\epsilon)$ for double precision flavors.

**err=2**

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \times \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\epsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

**err=3**

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} \times \text{slamch}(\epsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\epsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let $z=s \times (a \times \text{diag}(x))$, where $x$ is the solution for the current right-hand side and $s$ scales each row of $a \times \text{diag}(x)$ by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq nrhs$, and type of error $err$ is stored in `err_bnds_comp[(err-1) \times nrhs + i - 1]`.

**ipiv**

If `fact = 'N'` or `'E'`, then `ipiv` is an output argument and on exit contains the pivot indices from the factorization $A = P \times L \times U$ of the original matrix $A$ (if `fact = 'N'`) or of the equilibrated matrix $A$ (if `fact = 'E'`).
equed

If fact ≠ 'F', then equed is an output argument. It specifies the form of equilibration that was done (see the description of equed in Input Arguments section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

info

If info = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If info = -i, the i-th parameter had an illegal value.

If 0 < info ≤ n: Uinfo,info is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; rcond = 0 is returned.

If info = n+j: The solution corresponding to the j-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with k > j may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested params[2] = 0.0, then the j-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that err_bnds_norm[j - 1] = 0.0 or err_bnds_comp[j - 1] = 0.0. See the definition of err_bnds_norm and err_bnds_comp for err = 1. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp.

Return Values

This function returns a value info.

If info = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If info = -i, parameter i had an illegal value.

If 0 < info ≤ n: Uinfo,info is exactly zero. The factorization has been completed, but the factor U is exactly singular, so the solution and error bounds could not be computed; rcond = 0 is returned.

If info = n+j: The solution corresponding to the j-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides k with k > j may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested params[2] = 0.0, then the j-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest j such that for column major layout err_bnds_norm[j - 1] = 0.0 or err_bnds_comp[j - 1] = 0.0; or for row major layout err_bnds_norm[(j - 1)*n_err_bnds] = 0.0 or err_bnds_comp[(j - 1)*n_err_bnds] = 0.0). See the definition of err_bnds_norm and err_bnds_comp for err = 1. To get information about all of the right-hand sides, check err_bnds_norm or err_bnds_comp.

See Also

Matrix Storage Schemes

?gtsv

Computes the solution to the system of linear equations with a tridiagonal coefficient matrix A and multiple right-hand sides.

Syntax

lapack_int LAPACKE_sgtsv (int matrix_layout , lapack_int n , lapack_int nrhs , float * dl , float * d , float * du , float * b , lapack_int ldb );
lapack_int LAPACKE_dgtsv (int matrix_layout, lapack_int n, lapack_int nrhs, double * dl, double * d, double * du, double * b, lapack_int ldb);
lapack_int LAPACKE_cgtsv (int matrix_layout, lapack_int n, lapack_int nrhs, lapack_complex_float * dl, lapack_complex_float * d, lapack_complex_float * du, lapack_complex_float * b, lapack_int ldb);
lapack_int LAPACKE_zgtsv (int matrix_layout, lapack_int n, lapack_int nrhs, lapack_complex_double * dl, lapack_complex_double * d, lapack_complex_double * du, lapack_complex_double * b, lapack_int ldb);

Include Files
• mkl.h

Description
The routine solves for $X$ the system of linear equations $AX = B$, where $A$ is an $n$-by-$n$ tridiagonal matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions. The routine uses Gaussian elimination with partial pivoting.

Note that the equation $A^T X = B$ may be solved by interchanging the order of the arguments $du$ and $dl$.

Input Parameters

matrix_layout  
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

n  
The order of $A$, the number of rows in $B$; $n \geq 0$.

nrhs  
The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.

dl  
The array $dl$ (size $n - 1$) contains the $(n - 1)$ subdiagonal elements of $A$.

d  
The array $d$ (size $n$) contains the diagonal elements of $A$.

du  
The array $du$ (size $n - 1$) contains the $(n - 1)$ superdiagonal elements of $A$.

b  
The array $b$ of size $\max(1, ldb*nrhs)$ for column major layout and $\max(1, ldb*n)$ for row major layout contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

ldb  
The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

dl  
Overwritten by the $(n-2)$ elements of the second superdiagonal of the upper triangular matrix $U$ from the $LU$ factorization of $A$. These elements are stored in $dl[0]$, $\ldots$, $dl[n - 3]$.

d  
Overwritten by the $n$ diagonal elements of $U$.

du  
Overwritten by the $(n-1)$ elements of the first superdiagonal of $U$.

b  
Overwritten by the solution matrix $X$. 
Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, parameter i had an illegal value.

If info = i, $U_{ii}$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless i = n.

See Also

Matrix Storage Schemes

?gtsvx

Computes the solution to the real or complex system of linear equations with a tridiagonal coefficient matrix $A$ and multiple right-hand sides, and provides error bounds on the solution.

Syntax

```c
lapack_int LAPACKE_sgtsvx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, const float* dl, const float* d, const float* du, float* dlf, float* df, float* duf, float* du2, lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr);
```

```c
lapack_int LAPACKE_dgtsvx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, const double* dl, const double* d, const double* du, double* dlf, double* df, double* duf, double* du2, lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr);
```

```c
lapack_int LAPACKE_cgtsvx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_float* dl, const lapack_complex_float* d, const lapack_complex_float* du, lapack_complex_float* dlf, lapack_complex_float* df, lapack_complex_float* duf, lapack_complex_float* du2, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, lapack_complex_float* rcond, lapack_complex_float* ferr, lapack_complex_float* berr);
```

```c
lapack_int LAPACKE_zgtsvx( int matrix_layout, char fact, char trans, lapack_int n, lapack_int nrhs, const lapack_complex_double* dl, const lapack_complex_double* d, const lapack_complex_double* du, lapack_complex_double* dlf, lapack_complex_double* df, lapack_complex_double* duf, lapack_complex_double* du2, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, float* rcond, float* ferr, float* berr);
```

Include Files

- mkl.h

Description

The routine uses the LU factorization to compute the solution to a real or complex system of linear equations $A \times X = B$, $A^T \times X = B$, or $A^H \times X = B$, where $A$ is a tridiagonal matrix of order $n$, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?gtsvx performs the following steps:
1. If \texttt{fact = 'N'}, the \textit{LU} decomposition is used to factor the matrix \( A \) as \( A = LU \), where \( L \) is a product of permutation and unit lower bidiagonal matrices and \( U \) is an upper triangular matrix with nonzeros in only the main diagonal and first two superdiagonals.

2. If some \( U_{i,i} = 0 \), so that \( U \) is exactly singular, then the routine returns with \texttt{info = i}. Otherwise, the factored form of \( A \) is used to estimate the condition number of the matrix \( A \). If the reciprocal of the condition number is less than machine precision, \texttt{info = n + 1} is returned as a warning, but the routine still goes on to solve for \( X \) and compute error bounds as described below.

3. The system of equations is solved for \( X \) using the factored form of \( A \).

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

**Input Parameters**

\textit{matrix\_layout}

Specifies whether matrix storage layout is row major (\texttt{LAPACK\_ROW\_MAJOR}) or column major (\texttt{LAPACK\_COL\_MAJOR}).

\textit{fact}

Must be \texttt{'F'} or \texttt{'N'}.

Specifies whether or not the factored form of the matrix \( A \) has been supplied on entry.

If \texttt{fact = 'F'}: on entry, \( dlf, df, duf, du2 \), and \( ipiv \) contain the factored form of \( A \); arrays \( dl, d, du, dlf, df, duf, du2 \), and \( ipiv \) will not be modified.

If \texttt{fact = 'N'}, the matrix \( A \) will be copied to \( dlf, df \), and \( duf \) and factored.

\textit{trans}

Must be \texttt{'N'}, \texttt{'T'}, or \texttt{'C'}.

Specifies the form of the system of equations:

If \texttt{trans = 'N'}, the system has the form \( A \cdot X = B \) (No transpose).

If \texttt{trans = 'T'}, the system has the form \( A^T \cdot X = B \) (Transpose).

If \texttt{trans = 'C'}, the system has the form \( A^T \cdot X = B \) (Conjugate transpose).

\( n \)

The number of linear equations, the order of the matrix \( A \); \( n \geq 0 \).

\( nrhs \)

The number of right hand sides, the number of columns of the matrices \( B \) and \( X \); \( nrhs \geq 0 \).

\( dl, d, du, dlf, df, duf, du2, b \)

Arrays:

\( dl \), size \( (n - 1) \), contains the subdiagonal elements of \( A \).

\( d \), size \( (n) \), contains the diagonal elements of \( A \).

\( du \), size \( (n - 1) \), contains the superdiagonal elements of \( A \).

\( dlf \), size \( (n - 1) \). If \texttt{fact = 'F'}, then \( dlf \) is an input argument and on entry contains the \( (n - 1) \) multipliers that define the matrix \( L \) from the \textit{LU} factorization of \( A \) as computed by \texttt{?gttrf}.

\( df \), size \( (n) \). If \texttt{fact = 'F'}, then \( df \) is an input argument and on entry contains the \( n \) diagonal elements of the upper triangular matrix \( U \) from the \textit{LU} factorization of \( A \).

\( duf \), size \( (n - 1) \). If \texttt{fact = 'F'}, then \( duf \) is an input argument and on entry contains the \( (n - 1) \) elements of the first superdiagonal of \( U \).
du2, size \((n-2)\). If \(\text{fact} = 'F'\), then \(du2\) is an input argument and on entry contains the \((n-2)\) elements of the second superdiagonal of \(U\).

\(b\), size \(\max(ldb*nrhs)\) for column major layout and \(\max(ldb*n)\) for row major layout, contains the right-hand side matrix \(B\).

\(ldb\)

The leading dimension of \(b\); \(ldb \geq \max(1, n)\) for column major layout and \(ldb \geq nrhs\) for row major layout.

\(ldx\)

The leading dimension of \(x\); \(ldx \geq \max(1, n)\) for column major layout and \(ldx \geq nrhs\) for row major layout.

\(ipiv\)

Array, size at least \(\max(1, n)\). If \(\text{fact} = 'F'\), then \(ipiv\) is an input argument and on entry contains the pivot indices, as returned by ?gttrf.

**Output Parameters**

\(x\)

Array, size \(\max(1, ldx*nrhs)\) for column major layout and \(\max(1, ldx*n)\) for row major layout.

If \(\text{info} = 0\) or \(\text{info} = n+1\), the array \(x\) contains the solution matrix \(X\).

\(dlf\)

If \(\text{fact} = 'N'\), then \(dlf\) is an output argument and on exit contains the \((n-1)\) multipliers that define the matrix \(L\) from the \(LU\) factorization of \(A\).

\(df\)

If \(\text{fact} = 'N'\), then \(df\) is an output argument and on exit contains the \(n\) diagonal elements of the upper triangular matrix \(U\) from the \(LU\) factorization of \(A\).

\(duf\)

If \(\text{fact} = 'N'\), then \(duf\) is an output argument and on exit contains the \((n-1)\) elements of the first superdiagonal of \(U\).

\(du2\)

If \(\text{fact} = 'N'\), then \(du2\) is an output argument and on exit contains the \((n-2)\) elements of the second superdiagonal of \(U\).

\(ipiv\)

The array \(ipiv\) is an output argument if \(\text{fact} = 'N'\) and, on exit, contains the pivot indices from the factorization \(A = L*U\); row \(i\) of the matrix was interchanged with row \(ipiv[i-1]\). The value of \(ipiv[i-1]\) will always be \(i\) or \(i+1\); \(ipiv[i-1]=i\) indicates a row interchange was not required.

\(rcond\)

An estimate of the reciprocal condition number of the matrix \(A\). If \(rcond\) is less than the machine precision (in particular, if \(rcond = 0\)), the matrix is singular to working precision. This condition is indicated by a return code of \(\text{info}>0\).

\(ferr\)

Array, size at least \(\max(1, nrhs)\). Contains the estimated forward error bound for each solution vector \(x_j\) (the \(j\)-th column of the solution matrix \(X\)). If \(xtrue\) is the true solution corresponding to \(x_j\), \(ferr[j-1]\) is an estimated upper bound for the magnitude of the largest element in \(x_j - xtrue\) divided by the magnitude of the largest element in \(x_j\). The estimate is as reliable as the estimate for \(rcond\), and is almost always a slight overestimate of the true error.
Array, size at least \(\max(1, \text{nrhs})\). Contains the component-wise relative backward error for each solution vector \(x_j\), that is, the smallest relative change in any element of \(A\) or \(B\) that makes \(x_j\) an exact solution.

**Return Values**

This function returns a value \(\text{info}\).

- If \(\text{info} = 0\), the execution is successful.
- If \(\text{info} = -i\), parameter \(i\) had an illegal value.
- If \(\text{info} = i\), and \(i \leq n\), then \(U_i\) is exactly zero. The factorization has not been completed unless \(i = n\), but the factor \(U\) is exactly singular, so the solution and error bounds could not be computed; \(rcond = 0\) is returned.
- If \(\text{info} = i\), and \(i = n + 1\), then \(U\) is nonsingular, but \(rcond\) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \(rcond\) would suggest.

**See Also**

Matrix Storage Schemes

?dtsvb

*Computes the solution to the system of linear equations with a diagonally dominant tridiagonal coefficient matrix \(A\) and multiple right-hand sides.*

**Syntax**

void sdtsvb (const MKL_INT * \(n\), const MKL_INT * \(\text{nrhs}\), float * \(\text{dl}\), float * \(d\), const float * \(\text{du}\), float * \(b\), const MKL_INT * \(\text{ldb}\), MKL_INT * \(\text{info}\));

void ddtsvb (const MKL_INT * \(n\), const MKL_INT * \(\text{nrhs}\), double * \(\text{dl}\), double * \(d\), const double * \(\text{du}\), double * \(b\), const MKL_INT * \(\text{ldb}\), MKL_INT * \(\text{info}\));

void cdtsvb (const MKL_INT * \(n\), const MKL_INT * \(\text{nrhs}\), MKL_Complex8 * \(\text{dl}\), MKL_Complex8 * \(d\), const MKL_Complex8 * \(\text{du}\), MKL_Complex8 * \(b\), const MKL_INT * \(\text{ldb}\), MKL_INT * \(\text{info}\));

void zdtsvb (const MKL_INT * \(n\), const MKL_INT * \(\text{nrhs}\), MKL_Complex16 * \(\text{dl}\), MKL_Complex16 * \(d\), const MKL_Complex16 * \(\text{du}\), MKL_Complex16 * \(b\), const MKL_INT * \(\text{ldb}\), MKL_INT * \(\text{info}\));

**Include Files**

- mkl.h

**Description**

The ?dtsvb routine solves a system of linear equations \(A^TX = B\) for \(X\), where \(A\) is an \(n\)-by-\(n\) diagonally dominant tridiagonal matrix, the columns of matrix \(B\) are individual right-hand sides, and the columns of \(X\) are the corresponding solutions. The routine uses the BABE (Burning At Both Ends) algorithm.

Note that the equation \(A^TX = B\) may be solved by interchanging the order of the arguments \(\text{du}\) and \(\text{dl}\).

**Input Parameters**

\(n\) The order of \(A\), the number of rows in \(B\); \(n \geq 0\).
nrhs

The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.

dl, d, du, b

Arrays: $dl$ (size $n - 1$), $d$ (size $n$), $du$ (size $n - 1$), $b$(max($ldb*nrhs$) for column major layout and max($ldb*n$) for row major layout).

The array $dl$ contains the $(n - 1)$ subdiagonal elements of $A$.

The array $d$ contains the diagonal elements of $A$.

The array $du$ contains the $(n - 1)$ superdiagonal elements of $A$.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

ldb

The leading dimension of $b$; $ldb \geq \max(1, n)$.

Output Parameters

dl

Overwritten by the $(n-1)$ elements of the subdiagonal of the lower triangular matrices $L_1$, $L_2$ from the factorization of $A$ (see dttrfb).

d

Overwritten by the $n$ diagonal element reciprocals of $U$.

b

Overwritten by the solution matrix $X$.

info

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, $u_{ii}$ is exactly zero, and the solution has not been computed. The factorization has not been completed unless $i = n$.

Application Notes

A diagonally dominant tridiagonal system is defined such that $|d_i| > |dl_{i-1}| + |du_i|$ for any $i$:

$\forall 1 < i < n$, and $|d_1| > |du_1|, |d_n| > |dl_{n-1}|$

The underlying BABE algorithm is designed for diagonally dominant systems. Such systems have no numerical stability issue unlike the canonical systems that use elimination with partial pivoting (see ?gtsv).

The diagonally dominant systems are much faster than the canonical systems.

NOTE

- The current implementation of BABE has a potential accuracy issue on very small or large data close to the underflow or overflow threshold respectively. Scale the matrix before applying the solver in the case of such input data.
- Applying the ?dtsvb factorization to non-diagonally dominant systems may lead to an accuracy loss, or false singularity detected due to no pivoting.

?posv
Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix $A$ and multiple right-hand sides.
Syntax

```c
lapack_int LAPACKE_sposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
float * a, lapack_int lda, float * b, lapack_int ldb);

lapack_int LAPACKE_dposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
double * a, lapack_int lda, double * b, lapack_int ldb);

lapack_int LAPACKE_cposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);

lapack_int LAPACKE_dsposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
double * a, lapack_int lda, double * b, lapack_int ldb, double * x, lapack_int ldx,
lapack_int * iter);

lapack_int LAPACKE_zcposv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb,
lapack_complex_double * x, lapack_int ldx, lapack_int * iter);
```

Include Files
- mkl.h

Description

The routine solves for \( X \) the real or complex system of linear equations \( A \times X = B \), where \( A \) is an \( n \)-by-\( n \) symmetric/Hermitian positive-definite matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

The Cholesky decomposition is used to factor \( A \) as
\[
A = U^T \times U \quad \text{(real flavors)} \quad \text{and} \quad A = U^H \times U \quad \text{(complex flavors)}, \quad \text{if} \; \text{uplo} = 'U',
\]
or
\[
A = L \times L^T \quad \text{(real flavors)} \quad \text{and} \quad A = L \times L^H \quad \text{(complex flavors)}, \quad \text{if} \; \text{uplo} = 'L',
\]
where \( U \) is an upper triangular matrix and \( L \) is a lower triangular matrix. The factored form of \( A \) is then used to solve the system of equations \( A \times X = B \).

The \texttt{dsposv} and \texttt{zcposv} are mixed precision iterative refinement subroutines for exploiting fast single precision hardware. They first attempt to factorize the matrix in single precision (\texttt{dsposv}) or single complex precision (\texttt{zcposv}) and use this factorization within an iterative refinement procedure to produce a solution with double precision (\texttt{dsposv}) / double complex precision (\texttt{zcposv}) normwise backward error quality (see below). If the approach fails, the method switches to a double precision or double complex precision factorization respectively and computes the solution.

The iterative refinement is not going to be a winning strategy if the ratio single precision/complex performance over double precision/double complex performance is too small. A reasonable strategy should take the number of right-hand sides and the size of the matrix into account. This might be done with a call to \texttt{ilaenv} in the future. At present, iterative refinement is implemented.

The iterative refinement process is stopped if
\[ \text{iter} > \text{itermax} \]
or for all the right-hand sides:
\[
\text{rnmr} < \sqrt{n} \times \text{xnrm} \times \text{anrm} \times \text{eps} \times \text{bwdmax},
\]
where
- \text{iter} is the number of the current iteration in the iterative refinement process
- \text{rnmr} is the infinity-norm of the residual
• \(\text{nrm}\) is the infinity-norm of the solution
• \(\text{anrm}\) is the infinity-operator-norm of the matrix \(A\)
• \(\text{eps}\) is the machine epsilon returned by \text{dlamch(’Epsilon’)}.

The values \text{itermax} and \text{bwdmax} are fixed to 30 and 1.0d+00 respectively.

**Input Parameters**

\text{matrix\_layout} Specifies whether matrix storage layout is row major (\text{LAPACK\_ROW\_MAJOR}) or column major (\text{LAPACK\_COL\_MAJOR}).

\text{uplo} Must be ‘U’ or ‘L’.

Indicates whether the upper or lower triangular part of \(A\) is stored:
If \text{uplo} = ‘U’, the upper triangle of \(A\) is stored.
If \text{uplo} = ‘L’, the lower triangle of \(A\) is stored.

\(n\) The order of matrix \(A\); \(n \geq 0\).

\(\text{nrhs}\) The number of right-hand sides, the number of columns in \(B\); \(\text{nrhs} \geq 0\).

\(a, b\) Arrays: \(a(\text{size max}(1, lda))\), \(b, \text{size max}(ldb*\text{nrhs})\) for column major layout and \(\text{max}(ldb*n)\) for row major layout. The array \(a\) contains the upper or the lower triangular part of the matrix \(A\) (see \text{uplo}).

Note that in the case of \text{zcposv} the imaginary parts of the diagonal elements need not be set and are assumed to be zero.

The array \(b\) contains the matrix \(B\) whose columns are the right-hand sides for the systems of equations.

\(lda\) The leading dimension of \(a\); \(lda \geq \text{max}(1, n)\).

\(ldb\) The leading dimension of \(b\); \(ldb \geq \text{max}(1, n)\) for column major layout and \(ldb \geq \text{nrhs}\) for row major layout.

\(ldx\) The leading dimension of the array \(x\); \(ldx \geq \text{max}(1, n)\) for column major layout and \(ldx \geq \text{nrhs}\) for row major layout.

**Output Parameters**

\(a\) If \(\text{info} = 0\), the upper or lower triangular part of \(a\) is overwritten by the Cholesky factor \(U\) or \(L\), as specified by \text{uplo}.

If iterative refinement has been successfully used (\(\text{info} = 0\) and \(\text{iter} \geq 0\)), then \(A\) is unchanged.

If double precision factorization has been used (\(\text{info} = 0\) and \(\text{iter} < 0\)), then the array \(A\) contains the factors \(L\) or \(U\) from the Cholesky factorization.

Overwritten by the solution matrix \(X\).

\(b\)

Array, size \(\text{max}(1, lda*\text{nrhs})\) for column major layout and \(\text{max}(1, ldb*n)\) for row major layout. If \(\text{info} = 0\), contains the \(n\)-by-\(\text{nrhs}\) solution matrix \(X\).
iter

- If \( \text{iter} < 0 \): iterative refinement has failed, double precision factorization has been performed
- If \( \text{iter} = -1 \): the routine fell back to full precision for implementation- or machine-specific reason
- If \( \text{iter} = -2 \): narrowing the precision induced an overflow, the routine fell back to full precision
- If \( \text{iter} = -3 \): failure of \text{spotrf} for \text{dsposv}, or \text{cpotrf} for \text{zcposv}
- If \( \text{iter} = -31 \): stop the iterative refinement after the 30th iteration.

If \( \text{iter} > 0 \): iterative refinement has been successfully used. Returns the number of iterations.

Return Values

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), parameter \( i \) had an illegal value.
- If \( \text{info} = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive definite, so the factorization could not be completed, and the solution has not been computed.

See Also

Matrix Storage Schemes

?posvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix \( A \), and provides error bounds on the solution.

Syntax

```c
lapack_int LAPACKE_sposvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```c
lapack_int LAPACKE_dposvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

```c
lapack_int LAPACKE_cposvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af, lapack_int ldaf, char* equed, float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```

```c
lapack_int LAPACKE_zposvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af, lapack_int ldaf, char* equed, double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```
Include Files

- mkl.h

Description

The routine uses the Cholesky factorization $A=U^T U$ (real flavors) / $A=U^H U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where $A$ is a $n$-by-$n$ real symmetric/Hermitian positive definite matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?posvx performs the following steps:

1. If fact = 'E', real scaling factors $s$ are computed to equilibrate the system:
   $\text{diag}(s)*A*\text{diag}(s)*\text{inv}(\text{diag}(s))*X = \text{diag}(s)*B$.
   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(s)*A*\text{diag}(s)$ and $B$ by $\text{diag}(s)*B$.

2. If fact = 'N' or 'E', the Cholesky decomposition is used to factor the matrix $A$ (after equilibration if fact = 'E') as
   $A = U^T U$ (real), $A = U^H U$ (complex), if uplo = 'U',
   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if uplo = 'L',
   where $U$ is an upper triangular matrix and $L$ is a lower triangular matrix.

3. If the leading $i$-by-$i$ principal minor is not positive-definite, then the routine returns with info = $i$.
   Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, info = $n + 1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

$\text{matrix_layout}$

Specifies whether matrix storage layout is row major ($\text{LAPACK_ROW_MAJOR}$) or column major ($\text{LAPACK_COL_MAJOR}$).

$\text{fact}$

Must be 'F', 'N', or 'E'.

Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.

If fact = 'F': on entry, $af$ contains the factored form of $A$. If equed = 'Y', the matrix $A$ has been equilibrated with scaling factors given by $s$.

$a$ and $af$ will not be modified.

If fact = 'N', the matrix $A$ will be copied to $af$ and factored.

If fact = 'E', the matrix $A$ will be equilibrated if necessary, then copied to $af$ and factored.

$\text{uplo}$

Must be 'U' or 'L'.


Indicates whether the upper or lower triangular part of $A$ is stored:
If $uplo = 'U'$, the upper triangle of $A$ is stored.
If $uplo = 'L'$, the lower triangle of $A$ is stored.

$n$
The order of matrix $A; n \geq 0$.

$nrhs$
The number of right-hand sides, the number of columns in $B; nrhs \geq 0$.

$a$, $af$, $b$
Arrays: $a$(size max(1, $lda*n$)), $af$(size max(1, $ldaf*n$)), $b$, size max($ldb*nrhs$) for column major layout and max($ldb*n$) for row major layout, .
The array $a$ contains the matrix $A$ as specified by $uplo$. If $fact = 'F'$ and $equed = 'Y'$, then $A$ must have been equilibrated by the scaling factors in $s$, and $a$ must contain the equilibrated matrix $\text{diag}(s) \cdot A \cdot \text{diag}(s)$.
The array $af$ is an input argument if $fact = 'F'$. It contains the triangular factor $U$ or $L$ from the Cholesky factorization of $A$ in the same storage format as $A$. If $equed$ is not 'N', then $af$ is the factored form of the equilibrated matrix $\text{diag}(s) \cdot A \cdot \text{diag}(s)$.
The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$lda$
The leading dimension of $a; lda \geq \text{max}(1, n)$.

$ldaf$
The leading dimension of $af; ldaf \geq \text{max}(1, n)$.

$db$
The leading dimension of $b; ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

$equed$
Must be 'N' or 'Y'.
$equed$ is an input argument if $fact = 'F'$. It specifies the form of equilibration that was done:
if $equed = 'N'$, no equilibration was done (always true if $fact = 'N'$);
if $equed = 'Y'$, equilibration was done, that is, $A$ has been replaced by $\text{diag}(s) \cdot A \cdot \text{diag}(s)$.

$s$
Array, size ($n$). The array $s$ contains the scale factors for $A$. This array is an input argument if $fact = 'F'$ only; otherwise it is an output argument.
If $equed = 'N'$, $s$ is not accessed.
If $fact = 'F'$ and $equed = 'Y'$, each element of $s$ must be positive.

$ldx$
The leading dimension of the output array $x; ldx \geq \text{max}(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

**Output Parameters**

$x$
Array, size max(1, $ldx*nrhs$) for column major layout and max(1, $ldx*n$) for row major layout.
If $info = 0$ or $info = n+1$, the array $x$ contains the solution matrix $X$ to the original system of equations. Note that if $equed = 'Y$', $A$ and $B$ are modified on exit, and the solution to the equilibrated system is $\text{inv}(\text{diag}(s)) \times X$.

Array $a$ is not modified on exit if $fact = 'F'$ or '$N$', or if $fact = 'E'$ and $equed = 'N'$.

If $fact = 'E'$ and $equed = 'Y'$, $A$ is overwritten by $\text{diag}(s) \times A \times \text{diag}(s)$.

If $fact = 'N'$ or $'E'$, then $af$ is an output argument and on exit returns the triangular factor $U$ or $L$ from the Cholesky factorization $A=U^T \times U$ or $A=L \times L^T$ (real routines), $A=U^H \times U$ or $A=L \times L^H$ (complex routines) of the original matrix $A$ (if $fact = 'N'$), or of the equilibrated matrix $A$ (if $fact = 'E'$). See the description of $a$ for the form of the equilibrated matrix.

Overwritten by $\text{diag}(s) \times B$, if $equed = 'Y'$; not changed if $equed = 'N'$.

This array is an output argument if $fact \neq 'F'$. See the description of $s$ in Input Arguments section.

An estimate of the reciprocal condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

Array, size at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector $x_j$ (the $j$-th column of the solution matrix $X$). If $xtrue$ is the true solution corresponding to $x_j$, $ferr[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - xtrue)$ divided by the magnitude of the largest element in $x_j$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

Array, size at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

If $fact \neq 'F'$, then $equed$ is an output argument. It specifies the form of equilibration that was done (see the description of $equed$ in Input Arguments section).

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, and $i \leq n$, the leading minor of order $i$ (and therefore the matrix $A$ itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $rcond = 0$ is returned.
If \( info = i \) and \( i = n + 1 \), then \( U \) is nonsingular, but \( rcond \) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \( rcond \) would suggest.

See Also
Matrix Storage Schemes

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric or Hermitian positive-definite coefficient matrix \( A \) applying the Cholesky factorization.

Syntax

\[
\begin{align*}
\text{lapack_int LAPACK_dposvxx}( \text{int} \ matrix\_layout, \text{char} \ fact, \text{char} \ uplo, \text{lapack_int} \ n, \\
\text{lapack_int} \ nrhs, \text{float}^* \ a, \text{lapack_int} \ lda, \text{float}^* \ af, \text{lapack_int} \ ldaf, \text{char}^* \ equed, \\
\text{float}^* \ s, \text{float}^* \ b, \text{lapack_int} \ ldb, \text{float}^* \ x, \text{lapack_int} \ ldx, \text{float}^* \ rcond, \text{float}^* \\
\text{rpvgrw}, \text{float}^* \ berr, \text{lapack_int} \ n\_err\_bnds, \text{float}^* \err\_bnds\_norm, \text{float}^* \err\_bnds\_comp, \text{lapack_int} \ nparams, \text{const float}^* \params); \\
\text{lapack_int LAPACK_dposvxx}( \text{int} \ matrix\_layout, \text{char} \ fact, \text{char} \ uplo, \text{lapack_int} \ n, \\
\text{lapack_int} \ nrhs, \text{double}^* \ a, \text{lapack_int} \ lda, \text{double}^* \ af, \text{lapack_int} \ ldaf, \text{char}^* \ equed, \\
\text{double}^* \ s, \text{double}^* \ b, \text{lapack_int} \ ldb, \text{double}^* \ x, \text{lapack_int} \ ldx, \text{double}^* \ rcond, \text{double}^* \\
\text{rpvgrw}, \text{double}^* \ berr, \text{lapack_int} \ n\_err\_bnds, \text{double}^* \err\_bnds\_norm, \text{double}^* \err\_bnds\_comp, \text{lapack_int} \ nparams, \text{const double}^* \params); \\
\text{lapack_int LAPACK_cposvxx}( \text{int} \ matrix\_layout, \text{char} \ fact, \text{char} \ uplo, \text{lapack_int} \ n, \\
\text{lapack_int} \ nrhs, \text{lapack_complex_float}^* \ a, \text{lapack_int} \ lda, \text{lapack_complex_float}^* \ af, \text{lapack_int} \ ldaf, \text{char}^* \ equed, \\
\text{lapack_complex_float}^* \ x, \text{lapack_int} \ ldx, \text{lapack_complex_float}^* \ rcond, \text{lapack_complex_float}^* \\
\text{rpvgrw}, \text{lapack_complex_float}^* \ berr, \text{lapack_int} \ n\_err\_bnds, \text{lapack_complex_float}^* \err\_bnds\_norm, \text{lapack_complex_float}^* \err\_bnds\_comp, \text{lapack_int} \ nparams, \text{const float}^* \params); \\
\text{lapack_int LAPACK_zposvxx}( \text{int} \ matrix\_layout, \text{char} \ fact, \text{char} \ uplo, \text{lapack_int} \ n, \\
\text{lapack_int} \ nrhs, \text{lapack_complex_double}^* \ a, \text{lapack_int} \ lda, \text{lapack_complex_double}^* \ af, \text{lapack_int} \ ldaf, \text{char}^* \ equed, \\
\text{lapack_complex_double}^* \ x, \text{lapack_int} \ ldx, \text{lapack_complex_double}^* \ rcond, \text{lapack_complex_double}^* \\
\text{rpvgrw}, \text{lapack_complex_double}^* \ berr, \text{lapack_int} \ n\_err\_bnds, \text{lapack_complex_double}^* \err\_bnds\_norm, \text{lapack_complex_double}^* \err\_bnds\_comp, \text{lapack_int} \ nparams, \text{const double}^* \params); 
\end{align*}
\]

Include Files

- mkl.h

Description

The routine uses the Cholesky factorization \( A=U^T U \) (real flavors) / \( A=U^H U \) (complex flavors) or \( A=L^T L^T \) (real flavors) / \( A=L^H L^H \) (complex flavors) to compute the solution to a real or complex system of linear equations \( A^T X = B \), where \( A \) is an \( n \times n \) real symmetric/Hermitian positive definite matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions. Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error (\( O(\text{eps}) \), where \( \text{eps} \) is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.
The routine accepts user-provided factorizations and equilibration factors; see definitions of the \texttt{fact} and \texttt{equed} options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\varepsilon)$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine \texttt{?posvxx} performs the following steps:

1. If \texttt{fact} = 'E', scaling factors are computed to equilibrate the system:
   \[
   \text{diag}(s)^*A^*\text{diag}(s)^*\text{inv}(\text{diag}(s))^*X = \text{diag}(s)^*B
   \]
   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(s)^*A^*\text{diag}(s)$ and $B$ by $\text{diag}(s)^*B$.

2. If \texttt{fact} = 'N' or 'E', the Cholesky decomposition is used to factor the matrix $A$ (after equilibration if \texttt{fact} = 'E') as
   \[
   A = U^T * U \text{ (real), } A = U^H * U \text{ (complex), if } \texttt{uplo} = \text{'U'},
   \]
   or
   \[
   A = L * L^T \text{ (real), } A = L * L^H \text{ (complex), if } \texttt{uplo} = \text{'L'},
   \]
   where $U$ is an upper triangular matrix and $L$ is a lower triangular matrix.

3. If the leading $i$-by-$i$ principal minor is not positive-definite, the routine returns with \texttt{info} = $i$.
   Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$ (see the \texttt{rcond} parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for $X$ and compute error bounds.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. By default, unless \texttt{params[0]} is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

\textbf{Input Parameters}

\begin{itemize}
  \item \texttt{matrix_layout} Specifies whether matrix storage layout is row major (\texttt{LAPACK_ROW_MAJOR}) or column major (\texttt{LAPACK_COL_MAJOR}).
  \item \texttt{fact} Must be 'F', 'N', or 'E'.
    Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.
    If \texttt{fact} = 'F', on entry, \texttt{af} contains the factored form of $A$. If \texttt{equed} is not 'N', the matrix $A$ has been equilibrated with scaling factors given by $s$. Parameters $a$ and $af$ are not modified.
    If \texttt{fact} = 'N', the matrix $A$ will be copied to $af$ and factored.
    If \texttt{fact} = 'E', the matrix $A$ will be equilibrated, if necessary, copied to $af$ and factored.
  \item \texttt{uplo} Must be 'U' or 'L'.
    Indicates whether the upper or lower triangular part of $A$ is stored:
    If \texttt{uplo} = 'U', the upper triangle of $A$ is stored.
    If \texttt{uplo} = 'L', the lower triangle of $A$ is stored.
  \item \texttt{n} The number of linear equations; the order of the matrix $A$; $n \geq 0$.
\end{itemize}
nrhs

The number of right-hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$.

$a, af, b$

Arrays: $a$ (size $\max(\text{lda} \times n)$), $af$ (size $\max(\text{ldaf} \times n)$), $b$ (size $\max(1, \text{ldb} \times nrhs)$) for column major layout and $\max(1, \text{ldb} \times n)$ for row major layout.

The array $a$ contains the matrix $A$ as specified by $\text{uplo}$. If $\text{fact} = 'F'$ and $\text{equed} = 'Y'$, then $A$ must have been equilibrated by the scaling factors in $s$, and $a$ must contain the equilibrated matrix $\text{diag}(s) \times A \times \text{diag}(s)$.

The array $af$ is an input argument if $\text{fact} = 'F'$. It contains the triangular factor $U$ or $L$ from the Cholesky factorization of $A$ in the same storage format as $A$. If $\text{equed}$ is not 'N', then $af$ is the factored form of the equilibrated matrix $\text{diag}(s) \times A \times \text{diag}(s)$.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$\text{lda}$

The leading dimension of the array $a$; $\text{lda} \geq \max(1, n)$.

$\text{ldaf}$

The leading dimension of the array $af$; $\text{ldaf} \geq \max(1, n)$.

$\text{equed}$

Must be 'N' or 'Y'.

$\text{equed}$ is an input argument if $\text{fact} = 'F'$. It specifies the form of equilibration that was done:

If $\text{equed} = 'N'$, no equilibration was done (always true if $\text{fact} = 'N'$).

If $\text{equed} = 'Y'$, both row and column equilibration was done, that is, $A$ has been replaced by $\text{diag}(s) \times A \times \text{diag}(s)$.

$s$

Array, size $(n)$. The array $s$ contains the scale factors for $A$. This array is an input argument if $\text{fact} = 'F'$ only; otherwise it is an output argument.

If $\text{equed} = 'N'$, $s$ is not accessed.

If $\text{fact} = 'F'$ and $\text{equed} = 'Y'$, each element of $s$ must be positive. Each element of $s$ should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

$\text{ldb}$

The leading dimension of the array $b$; $\text{ldb} \geq \max(1, n)$ for column major layout and $\text{ldb} \geq nrhs$ for row major layout.

$\text{ldx}$

The leading dimension of the output array $x$; $\text{ldx} \geq \max(1, n)$ for column major layout and $\text{ldx} \geq nrhs$ for row major layout.

$n\_err\_bnds$

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See $err\_bnds\_norm$ and $err\_bnds\_comp$ descriptions in the Output Arguments section below.
Specifies the number of parameters set in params. If $\leq 0$, the params array is never referenced and default values are used.

Array, size max(1,$nparams$). Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the params argument.

params[0] : Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

=0.0 No refinement is performed and no error bounds are computed.

=1.0 Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

params[1] : Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in err_bnds_norm and err_bnds_comp may no longer be trustworthy.

params[2] : Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

**Output Parameters**

Array, size max(1, $ldx*nrhs$) for column major layout and max(1, $ldx*n$) for row major layout.

If $info = 0$, the array $x$ contains the solution $n$-by-$nrhs$ matrix $X$ to the original system of equations. Note that $A$ and $B$ are modified on exit if $equed\neq 'N'$, and the solution to the equilibrated system is:

$\text{inv(diag}(s)\text{)}*X$.

Array $a$ is not modified on exit if $fact = 'F'$ or $'N'$, or if $fact = 'E'$ and $equed = 'N'$.

If $fact = 'E'$ and $equed = 'Y'$, $A$ is overwritten by $\text{diag}(s)A\text{diag}(s)$. 
If `fact = 'N'` or `E'`, then `af` is an output argument and on exit returns the triangular factor `U` or `L` from the Cholesky factorization `A=U^T*U` or `A=L*L^T` (real routines), `A=U^H*U` or `A=L*L^H` (complex routines) of the original matrix `A` (if `fact = 'N'`), or of the equilibrated matrix `A` (if `fact = 'E'`). See the description of `a` for the form of the equilibrated matrix.

If `equed = 'N'`, `B` is not modified.

If `equed = 'Y'`, `B` is overwritten by `diag(s)*B`.

This array is an output argument if `fact='F'`. Each element of this array is a power of the radix. See the description of `s` in Input Arguments section.

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix `A` after equilibration (if done). If `rcond` is less than the machine precision, in particular, if `rcond = 0`, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

Contains the reciprocal pivot growth factor:

\[ \frac{\|A\|}{\|L\|} \]

If this is much less than 1, the stability of the `LU` factorization of the (equilibrated) matrix `A` could be poor. This also means that the solution `X`, estimated condition numbers, and error bounds could be unreliable. If factorization fails with `0 < info < n`, this parameter contains the reciprocal pivot growth factor for the leading `info` columns of `A`.

Array, size at least `max(1, nrhs)`. Contains the componentwise relative backward error for each solution vector `x_j`, that is, the smallest relative change in any element of `A` or `B` that makes `x_j` an exact solution.

Array of size `nrhs*n_err_bnds`. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the `i`-th solution vector

\[ \frac{ \max_j |x_{\text{true}}_{ij} - x_{ij}| }{ \max_j |x_{ij}| } \]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

\[ err=1 \]

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold `sqrt(n) * slamch(ε)` for single precision flavors and `sqrt(n) * dlamch(ε)` for double precision flavors.

\[ err=2 \]

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold `sqrt(n) * slamch(ε)`
for single precision flavors and
sqrt(n)*dlamch(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold
sqrt(n)*slamch(ε) for single precision flavors and sqrt(n)*dlamch(ε) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix Z are:

\[ \|Z\|_\infty \cdot \|Z^{-1}\|_\infty \]

Let \( z = s \cdot a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in \( \text{err_bnds_norm}[(\text{err}-1)\cdot\text{nrhs} + i - 1] \).

Array of size \( \text{nrhs} \cdot \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \( i \)-th solution vector:

\[
\max_j \left| \frac{X_{true,ij} - X_{ij}}{X_{true,ij}} \right|
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (params[2] = 0.0), then \( \text{err_bnds_comp} \) is not accessed.

err=1

"Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold sqrt(n)*slamch(ε) for single precision flavors and sqrt(n)*dlamch(ε) for double precision flavors.

err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold sqrt(n)*slamch(ε) for single precision flavors and sqrt(n)*dlamch(ε) for double precision flavors. This error bound should only be trusted if the previous boolean is true.
Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} \cdot \text{slamch}(e)$ for single precision flavors and $\sqrt{n} \cdot \text{dlamch}(e)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\| \cdot \|Z^{-1}\|$$

Let $z = s \cdot (a \cdot \text{diag}(x))$, where $x$ is the solution for the current right-hand side and $s$ scales each row of $a \cdot \text{diag}(x)$ by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error $\text{err}$ is stored in $\text{err_bnds_comp}[(\text{err}-1)\cdot\text{nrhs} + i - 1]$.

$\text{equed}$

If $\text{fact} \neq 'F'$, then $\text{equed}$ is an output argument. It specifies the form of equilibration that was done (see the description of $\text{equed}$ in Input Arguments section).

$\text{params}$

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

### Return Values

This function returns a value $\text{info}$.

If $\text{info} = 0$, the execution is successful. The solution to every right-hand side is guaranteed.

If $\text{info} = -i$, parameter $i$ had an illegal value.

If $0 < \text{info} < n$: $U_{\text{info},\text{info}}$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If $\text{info} = n+j$: The solution corresponding to the $j$-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides $k$ with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested $\text{params}[2] = 0.0$, then the $j$-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest $j$ such that for column major layout $\text{err_bnds_norm}[j-1] = 0.0$ or $\text{err_bnds_comp}[j-1] = 0.0$; or for row major layout $\text{err_bnds_norm}[(j-1)\cdot\text{n_err_bnds}] = 0.0$ or $\text{err_bnds_comp}[(j-1)\cdot\text{n_err_bnds}] = 0.0$).

See the definition of $\text{err_bnds_norm}$ and $\text{err_bnds_comp}$ for $\text{err} = 1$. To get information about all of the right-hand sides, check $\text{err_bnds_norm}$ or $\text{err_bnds_comp}$.

### See Also

Matrix Storage Schemes

$\text{ppsv}$

Computes the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix $A$ and multiple right-hand sides.
Syntax

```c
lapack_int LAPACKE_sppsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , float * ap , float * b , lapack_int ldb );
lapack_int LAPACKE_dppsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , double * ap , double * b , lapack_int ldb );
lapack_int LAPACKE_cppsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , lapack_complex_float * ap , lapack_complex_float * b , lapack_int ldb );
lapack_int LAPACKE_zppsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , lapack_complex_double * ap , lapack_complex_double * b , lapack_int ldb );
```

Include Files

- mkl.h

Description

The routine solves for $X$ the real or complex system of linear equations $A*X = B$, where $A$ is an $n$-by-$n$ real symmetric/Hermitian positive-definite matrix stored in packed format, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

The Cholesky decomposition is used to factor $A$ as

$$A = U^T * U$$ (real flavors) and $$A = U^H * U$$ (complex flavors), if uplo = 'U',

or

$$A = L * L^T$$ (real flavors) and $$A = L * L^H$$ (complex flavors), if uplo = 'L',

where $U$ is an upper triangular matrix and $L$ is a lower triangular matrix. The factored form of $A$ is then used to solve the system of equations $A*X = B$.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**: Must be 'U' or 'L'.

  Indicates whether the upper or lower triangular part of $A$ is stored:

  - If uplo = 'U', the upper triangle of $A$ is stored.
  - If uplo = 'L', the lower triangle of $A$ is stored.

- **n**: The order of matrix $A$; $n \geq 0$.

- **nrhs**: The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.

- **ap, b**: Arrays: $ap$ (size max(1,$n^*(n+1)/2$), $b$, size max($ldb*nrhs$) for column major layout and max($ldb*n$) for row major layout. The array $ap$ contains the upper or the lower triangular part of the matrix $A$ (as specified by uplo) in packed storage (see Matrix Storage Schemes).

  The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

- **ldb**: The leading dimension of $b$; $ldb \geq max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.
Output Parameters

\[ ap \]
If \( info = 0 \), the upper or lower triangular part of \( A \) in packed storage is overwritten by the Cholesky factor \( U \) or \( L \), as specified by \( uplo \).

\[ b \]
Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), parameter \( i \) had an illegal value.
If \( info = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

See Also

Matrix Storage Schemes

?ppsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite packed coefficient matrix \( A \), and provides error bounds on the solution.

Syntax

```c
lapack_int LAPACKE_sppsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, float* ap, float* afp, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_dppsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, double* ap, double* afp, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
lapack_int LAPACKE_cppsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float* ap, lapack_complex_float* afp, char* equed, float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
lapack_int LAPACKE_zppsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double* ap, lapack_complex_double* afp, char* equed, double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

Include Files

- mkl.h

Description

The routine uses the Cholesky factorization \( A = U^T U \) (real flavors) / \( A = U^H U \) (complex flavors) or \( A = L L^T \) (real flavors) / \( A = L L^H \) (complex flavors) to compute the solution to a real or complex system of linear equations \( A X = B \), where \( A \) is a \( n \)-by-\( n \) symmetric or Hermitian positive-definite matrix stored in packed format, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions. Error bounds on the solution and a condition estimate are also provided.
The routine \texttt{?ppsvx} performs the following steps:

1. If \texttt{fact} = 'E', real scaling factors \( s \) are computed to equilibrate the system:
   \[
   \text{diag}(s) \cdot A \cdot \text{diag}(s) \cdot \text{inv} \{ \text{diag}(s) \} \cdot X = \text{diag}(s) \cdot B.
   \]
   Whether or not the system will be equilibrated depends on the scaling of the matrix \( A \), but if equilibration is used, \( A \) is overwritten by \( \text{diag}(s) \cdot A \cdot \text{diag}(s) \) and \( B \) by \( \text{diag}(s) \cdot B \).

2. If \texttt{fact} = 'N' or 'E', the Cholesky decomposition is used to factor the matrix \( A \) (after equilibration if \texttt{fact} = 'E') as
   \[
   A = U^T \cdot U \quad \text{(real)}, \quad A = U^H \cdot U \quad \text{(complex)}, \quad \text{if uplo} = 'U',
   \]
   or
   \[
   A = L \cdot L^T \quad \text{(real)}, \quad A = L \cdot L^H \quad \text{(complex)}, \quad \text{if uplo} = 'L',
   \]
   where \( U \) is an upper triangular matrix and \( L \) is a lower triangular matrix.

3. If the leading \( i \)-by-\( i \) principal minor is not positive-definite, then the routine returns with \texttt{info} = \( i \). Otherwise, the factored form of \( A \) is used to estimate the condition number of the matrix \( A \). If the reciprocal of the condition number is less than machine precision, \texttt{info} = \( n+1 \) is returned as a warning, but the routine still goes on to solve for \( X \) and compute error bounds as described below.

4. The system of equations is solved for \( X \) using the factored form of \( A \).

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix \( X \) is premultiplied by \( \text{diag}(s) \) so that it solves the original system before equilibration.

\textbf{Input Parameters}

\begin{itemize}
  \item \texttt{matrix\_layout} \hspace{2cm} Specifies whether matrix storage layout is row major \texttt{(LAPACK\_ROW\_MAJOR)} or column major \texttt{(LAPACK\_COL\_MAJOR)}.
  \item \texttt{fact} \hspace{2cm} Must be 'F', 'N', or 'E'.
    Specifies whether or not the factored form of the matrix \( A \) is supplied on entry, and if not, whether the matrix \( A \) should be equilibrated before it is factored.
    If \texttt{fact} = 'F': on entry, \texttt{afp} contains the factored form of \( A \). If \texttt{equed} = 'Y', the matrix \( A \) has been equilibrated with scaling factors given by \texttt{s}.
    \texttt{ap} and \texttt{afp} will not be modified.
    If \texttt{fact} = 'N', the matrix \( A \) will be copied to \texttt{afp} and factored.
    If \texttt{fact} = 'E', the matrix \( A \) will be equilibrated if necessary, then copied to \texttt{afp} and factored.
  \item \texttt{uplo} \hspace{2cm} Must be 'U' or 'L'.
    Indicates whether the upper or lower triangular part of \( A \) is stored:
    If \texttt{uplo} = 'U', the upper triangle of \( A \) is stored.
    If \texttt{uplo} = 'L', the lower triangle of \( A \) is stored.
  \item \texttt{n} \hspace{2cm} The order of matrix \( A \); \( n \geq 0 \).
  \item \texttt{nrhs} \hspace{2cm} The number of right-hand sides; the number of columns in \( B \); \( nrhs \geq 0 \).
\end{itemize}
Arrays: (size max(1,\(n(n+1)/2\)), \(a fp\) (size max(1,\(n(n+1)/2\)), \(b\) of size max(1, \(ldb\*
rhs\)) for column major layout and max(1, \(ldb\*
\)) for row major layout.

The array \(ap\) contains the upper or lower triangle of the original symmetric/Hermitian matrix \(A\) in packed storage (see Matrix Storage Schemes). In case when \(fact = 'F'\) and \(equed = 'Y'\), \(ap\) must contain the equilibrated matrix \(\text{diag}(s)\*A\*\text{diag}(s)\).

The array \(a fp\) is an input argument if \(fact = 'F'\) and contains the triangular factor \(U\) or \(L\) from the Cholesky factorization of \(A\) in the same storage format as \(A\). If \(equed\) is not \('N'\), then \(a fp\) is the factored form of the equilibrated matrix \(A\).

The array \(b\) contains the matrix \(B\) whose columns are the right-hand sides for the systems of equations.

\(ldb\)

The leading dimension of \(b\); \(ldb \geq \max(1, n)\) for column major layout and \(ldb \geq \nrhs\) for row major layout.

\(equed\)

Must be \('N'\) or \('Y'\).

\(equed\) is an input argument if \(fact = 'F'\). It specifies the form of equilibration that was done:

if \(equed = 'N'\), no equilibration was done (always true if \(fact = 'N'\));

if \(equed = 'Y'\), equilibration was done, that is, \(A\) has been replaced by \(\text{diag}(s)\*A\*\text{diag}(s)\).

\(s\)

Array, size \((n)\). The array \(s\) contains the scale factors for \(A\). This array is an input argument if \(fact = 'F'\) only; otherwise it is an output argument.

If \(equed = 'N'\), \(s\) is not accessed.

If \(fact = 'F'\) and \(equed = 'Y'\), each element of \(s\) must be positive.

\(ldx\)

The leading dimension of the output array \(x\); \(ldx \geq \max(1, n)\) for column major layout and \(ldx \geq \nrhs\) for row major layout.

Output Parameters

\(x\)

Array, size max(1, \(ldx\*
rhs\)) for column major layout and max(1, \(ldx\*
\)) for row major layout.

If \(info = 0\) or \(info = n+1\), the array \(x\) contains the solution matrix \(X\) to the original system of equations. Note that if \(equed = 'Y'\), \(A\) and \(B\) are modified on exit, and the solution to the equilibrated system is \(\text{inv(diag}(s))\*X\).

\(ap\)

Array \(ap\) is not modified on exit if \(fact = 'F'\) or \('N'\), or if \(fact = 'E'\) and \(equed = 'N'\).

If \(fact = 'E'\) and \(equed = 'Y'\), \(ap\) is overwritten by \(\text{diag}(s)\*A\*\text{diag}(s)\).
If fact = 'N' or 'E', then afp is an output argument and on exit returns the triangular factor U or L from the Cholesky factorization $A = U^T U$ or $A = L L^T$ (real routines), $A = U^H U$ or $A = L L^H$ (complex routines) of the original matrix $A$ (if fact = 'N'), or of the equilibrated matrix $A$ (if fact = 'E'). See the description of ap for the form of the equilibrated matrix.

b

Overwritten by $\text{diag}(s) B$, if equed = 'Y'; not changed if equed = 'N'.

s

This array is an output argument if fact≠'F'. See the description of s in Input Arguments section.

rcond

An estimate of the reciprocal condition number of the matrix $A$ after equilibration (if done). If rcond is less than the machine precision (in particular, if $\text{rcond} = 0$), the matrix is singular to working precision. This condition is indicated by a return code of info > 0.

ferr

Array, size at least $\max(1, \text{nrhs})$. Contains the estimated forward error bound for each solution vector $x_j$ (the j-th column of the solution matrix $X$). If $\text{xtrue}$ is the true solution corresponding to $x_j$, $\text{ferr}[j-1]$ is an estimated upper bound for the magnitude of the largest element in $(x_j - \text{xtrue})$ divided by the magnitude of the largest element in $x_j$. The estimate is as reliable as the estimate for $\text{rcond}$, and is almost always a slight overestimate of the true error.

berr

Array, size at least $\max(1, \text{nrhs})$. Contains the component-wise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

equed

If fact≠'F', then equed is an output argument. It specifies the form of equilibration that was done (see the description of equed in Input Arguments section).

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, parameter i had an illegal value.

If info = i, and $i \leq n$, the leading minor of order i (and therefore the matrix $A$ itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; $\text{rcond} = 0$ is returned.

If info = i, and $i = n + 1$, then $U$ is nonsingular, but $\text{rcond}$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $\text{rcond}$ would suggest.

See Also

Matrix Storage Schemes
?pbsv
Computes the solution to the system of linear equations with a symmetric or Hermitian positive-definite band coefficient matrix \( A \) and multiple right-hand sides.

Syntax

```c
lapack_int LAPACKE_spbsv (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, float * ab, lapack_int ldab, float * b, lapack_int ldb);

lapack_int LAPACKE_dpbsv (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, double * ab, lapack_int ldab, double * b, lapack_int ldb);

lapack_int LAPACKE_cpbsv (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, lapack_complex_float * ab, lapack_int ldab, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zpbsv (int matrix_layout, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, lapack_complex_double * ab, lapack_int ldab, lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine solves for \( X \) the real or complex system of linear equations \( A \cdot X = B \), where \( A \) is an \( n \)-by-\( n \) symmetric/Hermitian positive definite band matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

The Cholesky decomposition is used to factor \( A \) as

\[
A = U^T \cdot U \quad (\text{real flavors}) \quad \text{and} \quad A = U^H \cdot U \quad (\text{complex flavors}), \quad \text{if} \quad uplo = 'U'
\]

or

\[
A = L \cdot L^T \quad (\text{real flavors}) \quad \text{and} \quad A = L \cdot L^H \quad (\text{complex flavors}), \quad \text{if} \quad uplo = 'L',
\]

where \( U \) is an upper triangular band matrix and \( L \) is a lower triangular band matrix, with the same number of superdiagonals or subdiagonals as \( A \). The factored form of \( A \) is then used to solve the system of equations \( A \cdot X = B \).

Input Parameters

- \texttt{matrix_layout}:
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \texttt{uplo}:
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of \( A \) is stored:
  If \texttt{uplo} = 'U', the upper triangle of \( A \) is stored.
  If \texttt{uplo} = 'L', the lower triangle of \( A \) is stored.
- \texttt{n}:
  The order of matrix \( A \); \( n \geq 0 \).
- \texttt{kd}:
  The number of superdiagonals of the matrix \( A \) if \texttt{uplo} = 'U', or the number of subdiagonals if \texttt{uplo} = 'L'; \( kd \geq 0 \).
nrhs

The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.

$ab$, $b$

Arrays: $ab$(size max(1, $ldab \times n$)), $b$ of size max(1, $ldb \times nrhs$) for column major layout and max(1, $ldb \times n$) for row major layout. The array $ab$ contains the upper or the lower triangular part of the matrix $A$ (as specified by $uplo$) in band storage (see Matrix Storage Schemes).

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$ldab$

The leading dimension of the array $ab$; $ldab \geq kd + 1$.

$ldb$

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

$ab$

The upper or lower triangular part of $A$ (in band storage) is overwritten by the Cholesky factor $U$ or $L$, as specified by $uplo$, in the same storage format as $A$.

$b$

Overwritten by the solution matrix $X$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, the leading minor of order $i$ (and therefore the matrix $A$ itself) is not positive-definite, so the factorization could not be completed, and the solution has not been computed.

See Also

Matrix Storage Schemes

?pbsvx

Uses the Cholesky factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive-definite band coefficient matrix $A$, and provides error bounds on the solution.

Syntax

```c
lapack_int LAPACKE_spbsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, float* ab, lapack_int ldab, float* afb, lapack_int ldafb, char* equed, float* s, float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dpbsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, double* ab, lapack_int ldab, double* afb, lapack_int ldafb, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```
lapack_int LAPACKE_cpbsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, lapack_complex_float* ab, lapack_int ldab, lapack_complex_float* afb, lapack_int ldafb, char* equed, float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zpbsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int kd, lapack_int nrhs, lapack_complex_double* ab, lapack_int ldab, lapack_complex_double* afb, lapack_int ldafb, char* equed, double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

Include Files
• mkl.h

Description
The routine uses the Cholesky factorization $A=U^T U$ (real flavors) / $A=U^H U$ (complex flavors) or $A=L*L^T$ (real flavors) / $A=L*L^H$ (complex flavors) to compute the solution to a real or complex system of linear equations $A*X = B$, where $A$ is a $n$-by-$n$ symmetric or Hermitian positive definite band matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?pbsvx performs the following steps:

1. If $\text{fact} = 'E'$, real scaling factors $s$ are computed to equilibrate the system:
   
   $\text{diag}(s) A \text{diag}(s) \text{inv(\text{diag}(s))} X = \text{diag}(s) B$

   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(s) A \text{diag}(s)$ and $B$ by $\text{diag}(s) B$.

2. If $\text{fact} = 'N'$ or $'E'$, the Cholesky decomposition is used to factor the matrix $A$ (after equilibration if $\text{fact} = 'E'$) as

   $A = U^T U$ (real), $A = U^H U$ (complex), if $\text{uplo} = 'U'$,

   or $A = L*L^T$ (real), $A = L*L^H$ (complex), if $\text{uplo} = 'L'$,

   where $U$ is an upper triangular band matrix and $L$ is a lower triangular band matrix.

3. If the leading $i$-by-$i$ principal minor is not positive definite, then the routine returns with $\text{info} = i$.

   Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, $\text{info} = n+1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(s)$ so that it solves the original system before equilibration.

Input Parameters

$\text{matrix_layout}$

  Specifies whether matrix storage layout is row major ($\text{LAPACK\_ROW\_MAJOR}$) or column major ($\text{LAPACK\_COL\_MAJOR}$).

$\text{fact}$

  Must be $'F'$, $'N'$, or $'E'$.
Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.

If $\text{fact} = 'F'$: on entry, $\text{afb}$ contains the factored form of $A$. If $\text{equed} = 'Y'$, the matrix $A$ has been equilibrated with scaling factors given by $s$.

$\text{ab}$ and $\text{afb}$ will not be modified.

If $\text{fact} = 'N'$, the matrix $A$ will be copied to $\text{afb}$ and factored.

If $\text{fact} = 'E'$, the matrix $A$ will be equilibrated if necessary, then copied to $\text{afb}$ and factored.

$\text{uplo}$
Indicates whether the upper or lower triangular part of $A$ is stored:
- If $\text{uplo} = 'U'$, the upper triangle of $A$ is stored.
- If $\text{uplo} = 'L'$, the lower triangle of $A$ is stored.

$n$
The order of matrix $A$; $n \geq 0$.

$kd$
The number of superdiagonals or subdiagonals in the matrix $A$; $kd \geq 0$.

$\text{nrhs}$
The number of right-hand sides, the number of columns in $B$; $\text{nrhs} \geq 0$.

$\text{ab}$, $\text{afb}$, $\text{b}$
Arrays: $\text{ab}\text{(size max(1, ldab*n))}$, $\text{afb}\text{(size max(1, ldafb*n))}$, $\text{b}\text{of size max(1, ldb*nrhs)}$ for column major layout and max(1, ldb*n) for row major layout.

The array $\text{ab}$ contains the upper or lower triangle of the matrix $A$ in band storage (see Matrix Storage Schemes).

If $\text{fact} = 'F'$ and $\text{equed} = 'Y'$, then $\text{ab}$ must contain the equilibrated matrix $\text{diag}(s)*A*\text{diag}(s)$.

The array $\text{afb}$ is an input argument if $\text{fact} = 'F'$. It contains the triangular factor $U$ or $L$ from the Cholesky factorization of the band matrix $A$ in the same storage format as $A$. If $\text{equed} = 'Y'$, then $\text{afb}$ is the factored form of the equilibrated matrix $A$.

The array $\text{b}$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$\text{ldab}$
The leading dimension of $\text{ab}$; $\text{ldab} \geq kd+1$.

$\text{ldafb}$
The leading dimension of $\text{afb}$; $\text{ldafb} \geq kd+1$.

$\text{ldb}$
The leading dimension of $\text{b}$; $\text{ldb} \geq \text{max(1, n)}$ for column major layout and $\text{ldb} \geq \text{nrhs}$ for row major layout.

$\text{equed}$
Must be 'N' or 'Y'.

$\text{equed}$ is an input argument if $\text{fact} = 'F'$. It specifies the form of equilibration that was done:
- If $\text{equed} = 'N'$, no equilibration was done (always true if $\text{fact} = 'N'$)
if equed = 'Y', equilibration was done, that is, A has been replaced by \( \text{diag}(s) \times A \times \text{diag}(s) \).

Array, size \((n)\). The array \(s\) contains the scale factors for \(A\). This array is an input argument if fact = 'F' only; otherwise it is an output argument.

If equed = 'N', \(s\) is not accessed.

If fact = 'F' and equed = 'Y', each element of \(s\) must be positive.

The leading dimension of the output array \(x\); \(ldx \geq \max(1, n)\) for column major layout and \(ldx \geq nrhs\) for row major layout.

**Output Parameters**

Array, size \(\max(1, ldx*nrhs)\) for column major layout and \(\max(1, ldx*n)\) for row major layout.

If info = 0 or info = \(n+1\), the array \(x\) contains the solution matrix \(X\) to the original system of equations. Note that if equed = 'Y', \(A\) and \(B\) are modified on exit, and the solution to the equilibrated system is \(\text{inv(\text{diag}(s)}) \times X\).

On exit, if fact = 'E' and equed = 'Y', \(A\) is overwritten by \(\text{diag}(s) \times A \times \text{diag}(s)\).

If fact = 'N' or 'E', then \(afb\) is an output argument and on exit returns the triangular factor \(U\) or \(L\) from the Cholesky factorization \(A=U^T \times U\) or \(A=L \times L^T\) (real routines), \(A=U^H \times U\) or \(A=L \times L^H\) (complex routines) of the original matrix \(A\) (if fact = 'N'), or of the equilibrated matrix \(A\) (if fact = 'E'). See the description of \(ab\) for the form of the equilibrated matrix.

Overwritten by \(\text{diag}(s) \times B\), if equed = 'Y'; not changed if equed = 'N'.

This array is an output argument if fact\#'F'. See the description of \(s\) in **Input Arguments** section.

An estimate of the reciprocal condition number of the matrix \(A\) after equilibration (if done). If rcond is less than the machine precision (in particular, if \(rcond = 0\)), the matrix is singular to working precision. This condition is indicated by a return code of info > 0.

Array, size at least \(\max(1, nrhs)\). Contains the estimated forward error bound for each solution vector \(x_j\) (the \(j\)-th column of the solution matrix \(X\)). If \(xtrue\) is the true solution corresponding to \(x_j\), \(ferr[j-1]\) is an estimated upper bound for the magnitude of the largest element in \((x_j - xtrue)\) divided by the magnitude of the largest element in \(x_j\). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.
berr

Array, size at least \( \max(1, nrhs) \). Contains the component-wise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of \( A \) or \( B \) that makes \( x_j \) an exact solution.

equed

If \texttt{fact}'F', then \texttt{equed} is an output argument. It specifies the form of equilibration that was done (see the description of \texttt{equed} in \textit{Input Arguments} section).

**Return Values**

This function returns a value \texttt{info}.

If \texttt{info} = 0, the execution is successful.

If \texttt{info} = -\texttt{i}, parameter \texttt{i} had an illegal value.

If \texttt{info} = \texttt{i}, and \( i \leq n \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed; \( rcond = 0 \) is returned. If \texttt{info} = \texttt{i}, and \( i = n + 1 \), then \( U \) is nonsingular, but \( rcond \) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \( rcond \) would suggest.

**See Also**

Matrix Storage Schemes

\texttt{?ptsv}

\textit{Computes the solution to the system of linear equations with a symmetric or Hermitian positive definite tridiagonal coefficient matrix \( A \) and multiple right-hand sides.}

**Syntax**

\begin{verbatim}
lapack_int LAPACKE_sptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d, float* e, float* b, lapack_int ldb );
lapack_int LAPACKE_dptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d, double* e, double* b, lapack_int ldb );
lapack_int LAPACKE_cptsv( int matrix_layout, lapack_int n, lapack_int nrhs, float* d, lapack_complex_float* e, lapack_complex_float* b, lapack_int ldb );
lapack_int LAPACKE_zptsv( int matrix_layout, lapack_int n, lapack_int nrhs, double* d, lapack_complex_double* e, lapack_complex_double* b, lapack_int ldb );
\end{verbatim}

**Include Files**

- mkl.h

**Description**

The routine solves for \( X \) the real or complex system of linear equations \( A^*X = B \), where \( A \) is an \( n \)-by-\( n \) symmetric/Hermitian positive-definite tridiagonal matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

\( A \) is factored as \( A = L*D*L^T \) (real flavors) or \( A = L*D*L^H \) (complex flavors), and the factored form of \( A \) is then used to solve the system of equations \( A^*X = B \).
Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\( n \)

The order of matrix \( A \); \( n \geq 0 \).

\( nrhs \)

The number of right-hand sides, the number of columns in \( B \); \( nrhs \geq 0 \).

d

Array, dimension at least \( \max(1, n) \). Contains the diagonal elements of the tridiagonal matrix \( A \).

e, b

Arrays: \( e \) (size \( n - 1 \)), \( b \) of size \( \max(1, ldb \times nrhs) \) for column major layout and \( \max(1, ldb \times n) \) for row major layout. The array \( e \) contains the \( (n - 1) \) subdiagonal elements of \( A \).

The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

\( ldb \)

The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Output Parameters

\( d \)

Overwritten by the \( n \) diagonal elements of the diagonal matrix \( D \) from the \( L^TDL^T \) (real)/ \( L^HDL^H \) (complex) factorization of \( A \).

\( e \)

Overwritten by the \( (n - 1) \) subdiagonal elements of the unit bidiagonal factor \( L \) from the factorization of \( A \).

\( b \)

Overwritten by the solution matrix \( X \).

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.

If \( info = i \), the leading minor of order \( i \) (and therefore the matrix \( A \) itself) is not positive-definite, and the solution has not been computed. The factorization has not been completed unless \( i = n \).

See Also

Matrix Storage Schemes

?ptsvx

Uses factorization to compute the solution to the system of linear equations with a symmetric (Hermitian) positive definite tridiagonal coefficient matrix \( A \), and provides error bounds on the solution.

Syntax

```c
lapack_int LAPACKE_sptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs,
const float* d, const float* e, float* df, float* ef, const float* b, lapack_int ldb,
float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```
lapack_int LAPACKE_dptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs, 
const double* d, const double* e, double* df, double* ef, const double* b, lapack_int 
ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs, 
const float* d, const lapack_complex_float* e, float* df, lapack_complex_float* ef, 
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, 
float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zptsvx( int matrix_layout, char fact, lapack_int n, lapack_int nrhs, 
const double* d, const lapack_complex_double* e, double* df, lapack_complex_double* ef, 
const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, 
double* rcond, double* ferr, double* berr );

Include Files

• mkl.h

Description

The routine uses the Cholesky factorization $A = L^TDL^T$ (real)/$A = L^HDL^H$ (complex) to compute the solution to a real or complex system of linear equations $AX = B$, where $A$ is a $n$-by-$n$ symmetric or Hermitian positive definite tridiagonal matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?ptsvx performs the following steps:

1. If $fact = 'N'$, the matrix $A$ is factored as $A = L^TDL^T$ (real flavors)/$A = L^HDL^H$ (complex flavors), where $L$ is a unit lower bidiagonal matrix and $D$ is diagonal. The factorization can also be regarded as having the form $A = U^TDL^T$ (real flavors)/$A = U^HL^H$ (complex flavors).
2. If the leading $i$-by-$i$ principal minor is not positive-definite, then the routine returns with $info = i$. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.
3. The system of equations is solved for $X$ using the factored form of $A$.
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

Input Parameters

$matrix_layout$  
Specifies whether matrix storage layout is row major ($LAPACK_ROW_MAJOR$) or column major ($LAPACK_COL_MAJOR$).

$fact$  
Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix $A$ is supplied on entry.

If $fact = 'F'$: on entry, $df$ and $ef$ contain the factored form of $A$. Arrays $d$, $e$, $df$, and $ef$ will not be modified.

If $fact = 'N'$, the matrix $A$ will be copied to $df$ and $ef$, and factored.

$n$  
The order of matrix $A$; $n \geq 0$.

$nrhs$  
The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$. 
Arrays: $d$ (size n), $df$ (size n).
The array $d$ contains the $n$ diagonal elements of the tridiagonal matrix $A$.
The array $df$ is an input argument if $fact = 'F'$ and on entry contains the $n$ diagonal elements of the diagonal matrix $D$ from the $L^TDL^T$ (real)/$L^HDL^H$ (complex) factorization of $A$.

Arrays: $e$ (size n-1), $ef$ (size n-1), $b$, size max($ldb*nrhs$) for column major layout and max($ldb*n$) for row major layout. The array $e$ contains the $(n-1)$ subdiagonal elements of the tridiagonal matrix $A$.
The array $ef$ is an input argument if $fact = 'F'$ and on entry contains the $(n-1)$ subdiagonal elements of the unit bidiagonal factor $L$ from the $L^TDL^T$ (real)/$L^HDL^H$ (complex) factorization of $A$.
The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$ldb$
The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq \max(1, nrhs)$ for row major layout.

$ldx$
The leading dimension of $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq \max(1, nrhs)$ for row major layout.

Output Parameters

$x$
Array, size $\max(1, ldx*nrhs)$ for column major layout and $\max(1, ldx*n)$ for row major layout.
If $info = 0$ or $info = n+1$, the array $x$ contains the solution matrix $X$ to the system of equations.

$df$, $ef$
These arrays are output arguments if $fact = 'N'$. See the description of $df$, $ef$ in Input Arguments section.

$rcond$
An estimate of the reciprocal condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

$ferr$
Array, size at least $\max(1, nrhs)$. Contains the estimated forward error bound for each solution vector $x_j$ (the $j$-th column of the solution matrix $X$). If $xtrue$ is the true solution corresponding to $x_j$, $ferr_j$ is an estimated upper bound for the magnitude of the largest element in $(x_j - xtrue)$ divided by the magnitude of the largest element in $x_j$. The estimate is as reliable as the estimate for $rcond$, and is almost always a slight overestimate of the true error.

$berr$
Array, size at least $\max(1, nrhs)$. Contains the component-wise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.
Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, parameter i had an illegal value.

If info = i, and i≤n, the leading minor of order i (and therefore the matrix A itself) is not positive-definite, so the factorization could not be completed, and the solution and error bounds could not be computed; rcond =0 is returned.

If info = i, and i = n + 1, then U is nonsingular, but rcond is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of rcond would suggest.

See Also
Matrix Storage Schemes

?sysv
Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

lapack_int LAPACKE_ssysv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, float * a, lapack_int lda, lapack_int * ipiv, float * b, lapack_int ldb);

lapack_int LAPACKE_dsysv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, double * a, lapack_int lda, lapack_int * ipiv, double * b, lapack_int ldb);

lapack_int LAPACKE_csysv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zsysv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * a, lapack_int lda, lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);

Include Files

• mkl.h

Description

The routine solves for X the real or complex system of linear equations A*X = B, where A is an n-by-n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

The diagonal pivoting method is used to factor A as A = U*D*U^T or A = L*D*L^T, where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and D is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of A is then used to solve the system of equations A*X = B.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of A is stored:
  - If uplo = 'U', the upper triangle of A is stored.
  - If uplo = 'L', the lower triangle of A is stored.

- **n**
  The order of matrix A; n ≥ 0.

- **nrhs**
  The number of right-hand sides; the number of columns in B; nrhs ≥ 0.

- **a, b**
  Arrays: a(size max(1, lda*n)), b of size max(1, ldb*nrhs) for column major layout and max(1, ldb*n) for row major layout.
  The array a contains the upper or the lower triangular part of the symmetric matrix A (see uplo).
  The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

- **lda**
  The leading dimension of a; lda ≥ max(1, n).

- **ldb**
  The leading dimension of b; ldb ≥ max(1, n) for column major layout and ldb≥nrhs for row major layout.

**Output Parameters**

- **a**
  If info = 0, a is overwritten by the block-diagonal matrix D and the multipliers used to obtain the factor U (or L) from the factorization of A as computed by ?sytrf.

- **b**
  If info = 0, b is overwritten by the solution matrix X.

- **ipiv**
  Array, size at least max(1, n). Contains details of the interchanges and the block structure of D, as determined by ?sytrf.
  If ipiv[i-1] = k >0, then d_{ii} is a 1-by-1 diagonal block, and the i-th row and column of A was interchanged with the k-th row and column.
  If uplo = 'U' and ipiv[i] = ipiv[i-1] = -m < 0, then D has a 2-by-2 block in rows/columns i and i+1, and (i)-th row and column of A was interchanged with the m-th row and column.
  If uplo = 'L' and ipiv[i] = ipiv[i-1] = -m < 0, then D has a 2-by-2 block in rows/columns i and i+1, and (i+1)-th row and column of A was interchanged with the m-th row and column.

**Return Values**

This function returns a value info.

If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.

If info = i, d_{ii} is 0. The factorization has been completed, but D is exactly singular, so the solution could not be computed.

See Also
Matrix Storage Schemes

?sysv_aa
Computes the solution to a system of linear equations
A * X = B for symmetric matrices.

```c
lapack_int LAPACK_ssysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, float * A, lapack_int lda, lapack_int * ipiv, float * B, lapack_int ldb);

lapack_int LAPACK_dsysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, double * A, lapack_int lda, lapack_int * ipiv, double * B, lapack_int ldb);

lapack_int LAPACK_csysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * A, lapack_int lda, lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);

lapack_int LAPACK_zsysv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * A, lapack_int lda, lapack_int * ipiv, lapack_complex_double * B, lapack_int ldb);
```

Description
The ?sysv routine computes the solution to a complex system of linear equations A * X = B, where A is an n-by-n symmetric matrix and X and B are n-by-nrhs matrices.

Aasen's algorithm is used to factor A as A = U * T * U^T, if uplo = 'U', or A = L * T * L^T, if uplo = 'L', where U (or L) is a product of permutation and unit upper (lower) triangular matrices, and T is symmetric tridiagonal. The factored form of A is then used to solve the system of equations A * X = B.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  - = 'U': The upper triangle of A is stored.
  - = 'L': The lower triangle of A is stored.

- **n**
  The number of linear equations; that is, the order of the matrix A. n ≥ 0.

- **nrhs**
  The number of right-hand sides; that is, the number of columns of the matrix B. nrhs ≥ 0.

- **A**
  Array of size max(1, lda*n). On entry, the symmetric matrix A. If uplo = 'U', the leading n-by-n upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If uplo = 'L', the leading n-by-n lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

- **lda**
  The leading dimension of the array A.
Array of size max(1, ldb*nrhs) for column-major layout and max(1, ldb*n) for row-major layout. On entry, the n-by-nrhs right-hand side matrix B.

The leading dimension of the array B. ldb ≥ max(1, n) for column-major layout and ldb ≥ nrhs for row-major layout.

On exit, if info = 0, the tridiagonal matrix T and the multipliers used to obtain the factor U or L from the factorization A = U*T*Uᵀ or A = L*T*Lᵀ as computed by ?sytrf.

Array of size n. On exit, it contains the details of the interchanges; that is, the row and column k of A were interchanged with the row and column ipiv(k).

On exit, if info = 0, the n-by-nrhs solution matrix X.

This function returns a value info.
= 0: Successful exit.
< 0: If info = -i, the i-th argument had an illegal value.
> 0: If info = i, D(i,i) is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution could not be computed.

Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix A and multiple right-hand sides.

Syntax

```c
lapack_int LAPACKE_ssysv_rook (int matrix_layout, char uplo , lapack_int n ,
lapack_int nrhs , float * a , lapack_int lda , lapack_int * ipiv , float * b ,
lapack_int ldb );
```

```c
lapack_int LAPACKE_dsysv_rook (int matrix_layout, char uplo , lapack_int n ,
lapack_int nrhs , double * a , lapack_int lda , lapack_int * ipiv , double * b ,
lapack_int ldb );
```

```c
lapack_int LAPACKE_csysv_rook (int matrix_layout, char uplo , lapack_int n ,
lapack_int nrhs , lapack_complex_float * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_float * b , lapack_int ldb );
```

```c
lapack_int LAPACKE_zsysv_rook (int matrix_layout, char uplo , lapack_int n ,
lapack_int nrhs , lapack_complex_double * a , lapack_int lda , lapack_int * ipiv ,
lapack_complex_double * b , lapack_int ldb );
```

Include Files
- mkl.h

Description
The routine solves for \( X \) the real or complex system of linear equations \( A*X = B \), where \( A \) is an \( n \)-by-\( n \) symmetric matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

The diagonal pivoting method is used to factor \( A \) as \( A = U*D*U^T \) or \( A = L*D*L^T \), where \( U \) (or \( L \)) is a product of permutation and unit upper (lower) triangular matrices, and \( D \) is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The \(?sysv_rook\) routine is called to compute the factorization of a complex symmetric matrix \( A \) using the bounded Bunch-Kaufman ("rook") diagonal pivoting method.

The factored form of \( A \) is then used to solve the system of equations \( A*X = B \).

**Input Parameters**

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  - Must be 'U' or 'L'.
  - Indicates whether the upper or lower triangular part of \( A \) is stored:
    - If \( uplo = 'U' \), the upper triangle of \( A \) is stored.
    - If \( uplo = 'L' \), the lower triangle of \( A \) is stored.

- **n**
  - The order of matrix \( A \); \( n \geq 0 \).

- **nrhs**
  - The number of right-hand sides; the number of columns in \( B \); \( nrhs \geq 0 \).

- **a, b**
  - Arrays: \( a(\text{size max}(1, lda*n)) \), \( b(\text{size max}(1, ldb*nrhs)) \) for column major layout and \( \text{max}(1, lda*n) \) for row major layout.
  - The array \( a \) contains the upper or the lower triangular part of the symmetric matrix \( A \) (see \( uplo \)). The second dimension of \( a \) must be at least \( \text{max}(1, n) \).
  - The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations. The second dimension of \( b \) must be at least \( \text{max}(1, nrhs) \).

- **lda**
  - The leading dimension of \( a \); \( lda \geq \text{max}(1, n) \).

- **ldb**
  - The leading dimension of \( b \); \( ldb \geq \text{max}(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

**Output Parameters**

- **a**
  - If \( info = 0 \), \( a \) is overwritten by the block-diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) (or \( L \)) from the factorization of \( A \).

- **b**
  - If \( info = 0 \), \( b \) is overwritten by the solution matrix \( X \).

- **ipiv**
  - Array, size at least \( \text{max}(1, n) \). Contains details of the interchanges and the block structure of \( D \).
  - If \( ipiv[k-1] > 0 \), then rows and columns \( k \) and \( ipiv[k-1] \) were interchanged and \( D_{k,k} \) is a 1-by-1 diagonal block.
If uplo = 'U' and ipiv[k - 1] < 0 and ipiv[k - 2] < 0, then rows and columns k and \(-ipiv[k - 1]\) were interchanged, rows and columns \(k - 1\) and \(-ipiv[k - 2]\) were interchanged, and \(D_{k-1:k, k-1:k}\) is a 2-by-2 diagonal block.

If uplo = 'L' and ipiv[k - 1] < 0 and ipiv[k] < 0, then rows and columns k and \(-ipiv[k - 1]\) were interchanged, rows and columns \(k + 1\) and \(-ipiv[k]\) were interchanged, and \(D_{k:k+1, k:k+1}\) is a 2-by-2 diagonal block.

Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
If info = i, \(d_{ii}\) is 0. The factorization has been completed, but \(D\) is exactly singular, so the solution could not be computed.

See Also
Matrix Storage Schemes

?syrk
Computes the solution to system of linear equations \(A * X = B\) for SY matrices.

\[
lapack_int \text{LAPACKE_ssysv_rk}(int \text{matrix_layout}, \text{char} \ uplo, \text{lapack_int} \ n, \text{lapack_int} \ nrhs, \text{float} * \ A, \text{lapack_int} \ lda, \text{float} * \ e, \text{lapack_int} * \ ipiv, \text{float} * \ B, \text{lapack_int} \ ldb);
\]

\[
lapack_int \text{LAPACKE_dsysv_rk}(int \text{matrix_layout}, \text{char} \ uplo, \text{lapack_int} \ n, \text{lapack_int} \ nrhs, \text{double} * \ A, \text{lapack_int} \ lda, \text{double} * \ e, \text{lapack_int} * \ ipiv, \text{double} * \ B, \text{lapack_int} \ ldb);
\]

\[
lapack_int \text{LAPACKE_csysv_rk}(int \text{matrix_layout}, \text{char} \ uplo, \text{lapack_int} \ n, \text{lapack_int} \ nrhs, \text{lapack_complex_float} * \ A, \text{lapack_int} \ lda, \text{lapack_complex_float} * \ e, \text{lapack_int} * \ ipiv, \text{lapack_complex_float} * \ B, \text{lapack_int} \ ldb);
\]

\[
lapack_int \text{LAPACKE_zsysv_rk}(int \text{matrix_layout}, \text{char} \ uplo, \text{lapack_int} \ n, \text{lapack_int} \ nrhs, \text{lapack_complex_double} * \ A, \text{lapack_int} \ lda, \text{lapack_complex_double} * \ e, \text{lapack_int} * \ ipiv, \text{lapack_complex_double} * \ B, \text{lapack_int} \ ldb);
\]

Description
?syrk computes the solution to a real or complex system of linear equations \(A * X = B\), where \(A\) is an \(n\times n\) symmetric matrix and \(X\) and \(B\) are \(n\times nrhs\) matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor \(A\) as \(A = P*U*\text{Diag}(U^T)*(P^T)\), if \(uplo = 'U'\), or \(A = P*L*\text{Diag}(L^T)*(P^T)\), if \(uplo = 'L'\), where \(U\) (or \(L\)) is unit upper (or lower) triangular matrix, \(U^T\) (or \(L^T\)) is the transpose of \(U\) (or \(L\)), \(P\) is a permutation matrix, \(P^T\) is the transpose of \(P\), and \(D\) is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?sytrf_rk is called to compute the factorization of a real or complex symmetric matrix. The factored form of \(A\) is then used to solve the system of equations \(A * X = B\) by calling BLAS3 routine ?sytrs_3.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **uplo**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is stored:
  - `'U'`: The upper triangle of A is stored.
  - `'L'`: The lower triangle of A is stored.

- **n**
  The number of linear equations; that is, the order of the matrix A. \( n \geq 0 \).

- **nrhs**
  The number of right-hand sides; that is, the number of columns of the matrix B. \( nrhs \geq 0 \).

- **A**
  Array of size \( \max(1, lda*n) \). On entry, the symmetric matrix A. If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

- **lda**
  The leading dimension of the array A.

- **B**
  Array of size \( \max(1, ldb*nrhs) \). On entry, the \( n \)-by-\( nrhs \) right-hand side matrix B.

- **ldb**
  The leading dimension of the array B. \( ldb \geq \max(1, n) \) for column-major layout and \( ldb \geq nrhs \) for row-major layout.

**Output Parameters**

- **A**
  On exit, if \( info = 0 \), the diagonal of the block diagonal matrix D and factors U or L as computed by ?sytrf_rk:
  - *Only* diagonal elements of the symmetric block diagonal matrix D on the diagonal of A; that is, \( D(k,k) = A(k,k) \). Superdiagonal (or subdiagonal) elements of D are stored on exit in array e.
  - If \( uplo = 'U' \), factor U in the superdiagonal part of A. If \( uplo = 'L' \), factor L in the subdiagonal part of A. For more information, see the description of the ?sytrf_rk routine.

- **e**
  Array of size \( n \). On exit, contains the output computed by the factorization routine ?sytrf_rk; that is, the superdiagonal (or subdiagonal) elements of the symmetric block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks. If \( uplo = 'U' \), \( e(i) = D(i-1,i), i=1:N-1 \), and \( e(1) \) is set to 0. If \( uplo = 'L' \), \( e(i) = D(i+1,i), i=1:N-1 \), and \( e(n) \) is set to 0.

**NOTE** For 1-by-1 diagonal block \( D(k) \), where \( 1 \leq k \leq n \), the element \( e(k) \) is set to 0 in both the \( uplo = 'U' \) and \( uplo = 'L' \) cases. For more information, see the description of the ?sytrf_rk routine.
ipiv

Array of size n. Details of the interchanges and the block structure of D, as determined by ?sytrf_rk. For more information, see the description of the ?sytrf_rk routine.

B

On exit, if info = 0, the n-by-nrhs solution matrix X.

Return Values

This function returns a value info.

= 0: Successful exit.
< 0: If info = -k, the kth argument had an illegal value.
> 0: If info = k, the matrix A is singular. If uplo = 'U', column k in the upper triangular part of A contains all zeros. If uplo = 'L', column k in the lower triangular part of A contains all zeros. Therefore D(k,k) is exactly zero, and superdiagonal elements of column k of U (or subdiagonal elements of column k of L) are all zeros. The factorization has been completed, but the block diagonal matrix D is exactly singular, and division by zero will occur if it is used to solve a system of equations.

?sysvx

Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix A, and provides error bounds on the solution.

Syntax

lapack_int LAPACKE_ssysvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, const float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv, const float* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dsysvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, const double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int* ipiv, const double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_csysvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float* af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zsysvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double* af, lapack_int ldaf, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

Include Files

• mkl.h

Description

The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations A*X = B, where A is a n-by-n symmetric matrix, the columns of matrix B are individual right-hand sides, and the columns of X are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.
The routine \texttt{sysvx} performs the following steps:

1. If \texttt{fact} = 'N', the diagonal pivoting method is used to factor the matrix \( A \). The form of the factorization is \( A = U*D*U^T \) or \( A = L*D*L^T \), where \( U \) (or \( L \)) is a product of permutation and unit upper (lower) triangular matrices, and \( D \) is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some \( d_{ii} = 0 \), so that \( D \) is exactly singular, then the routine returns with \( \text{info} = i \). Otherwise, the factored form of \( A \) is used to estimate the condition number of the matrix \( A \). If the reciprocal of the condition number is less than machine precision, \( \text{info} = n+1 \) is returned as a warning, but the routine still goes on to solve for \( X \) and compute error bounds as described below.

3. The system of equations is solved for \( X \) using the factored form of \( A \).

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

**Input Parameters**

\texttt{matrix\_layout} 

Specifies whether matrix storage layout is row major (\texttt{LAPACK\_ROW\_MAJOR}) or column major (\texttt{LAPACK\_COL\_MAJOR}).

\texttt{fact}

Must be 'F' or 'N'.

\texttt{uplo}

Must be 'U' or 'L'.

\texttt{n}

The order of matrix \( A \); \( n \geq 0 \).

\texttt{nrhs}

The number of right-hand sides, the number of columns in \( B \); \( nrhs \geq 0 \).

\texttt{a, af, b}

Arrays: \( a(\text{size max}(1, lda*n)) \), \( af(\text{size max}(1, ldaf*n)) \), \( b(\text{size max}(1, ldb*nrhs)) \) for column major layout and \( \text{max}(1, ldb*n) \) for row major layout.

The array \( a \) contains the upper or the lower triangular part of the symmetric matrix \( A \) (see \texttt{uplo}).

The array \( af \) is an input argument if \texttt{fact} = 'F'. It contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) from the factorization \( A = U*D*U^T \) or \( A = L*D*L^T \) as computed by \texttt{?sytrf}.

The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

\texttt{lda}

The leading dimension of \( a \); \( lda \geq \text{max}(1, n) \).

\texttt{ldaf}

The leading dimension of \( af \); \( ldaf \geq \text{max}(1, n) \).
The leading dimension of \( b \); \( ldb \geq \max(1, n) \) for column major layout and \( ldb \geq nrhs \) for row major layout.

Array, size at least \( \max(1, n) \). The array \( ipiv \) is an input argument if \( fact = 'F' \). It contains details of the interchanges and the block structure of \( D \), as determined by \( ?sytrf \).

If \( ipiv[i-1] = k > 0 \), then \( d_{ii} \) is a 1-by-1 diagonal block, and the \( i \)-th row and column of \( A \) was interchanged with the \( k \)-th row and column.

If \( uplo = 'U' \) and \( ipiv[i] = ipiv[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \( (i) \)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

If \( uplo = 'L' \) and \( ipiv[i] = ipiv[i-1] = -m < 0 \), then \( D \) has a 2-by-2 block in rows/columns \( i \) and \( i+1 \), and \( (i+1) \)-th row and column of \( A \) was interchanged with the \( m \)-th row and column.

The leading dimension of the output array \( x \); \( ldx \geq \max(1, n) \) for column major layout and \( ldx \geq nrhs \) for row major layout.

Output Parameters

\( x \)

Array, size \( \max(1, ldx*nrhs) \) for column major layout and \( \max(1, ldx*n) \) for row major layout.

If \( info = 0 \) or \( info = n+1 \), the array \( x \) contains the solution matrix \( X \) to the system of equations.

\( af, ipiv \)

These arrays are output arguments if \( fact = 'N' \).

See the description of \( af, ipiv \) in Input Arguments section.

\( rcond \)

An estimate of the reciprocal condition number of the matrix \( A \). If \( rcond \) is less than the machine precision (in particular, if \( rcond = 0 \)), the matrix is singular to working precision. This condition is indicated by a return code of \( info > 0 \).

\( ferr \)

Array, size at least \( \max(1, nrhs) \). Contains the estimated forward error bound for each solution vector \( x_j \) (the \( j \)-th column of the solution matrix \( X \)). If \( xtrue \) is the true solution corresponding to \( x_j \), \( ferr[j-1] \) is an estimated upper bound for the magnitude of the largest element in \( (x_j - xtrue) \) divided by the magnitude of the largest element in \( x_j \). The estimate is as reliable as the estimate for \( rcond \), and is almost always a slight overestimate of the true error.

\( berr \)

Array, size at least \( \max(1, nrhs) \). Contains the component-wise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of \( A \) or \( B \) that makes \( x_j \) an exact solution.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), parameter \( i \) had an illegal value.
If \( info = i \), and \( i \leq n \), then \( d_{ii} \) is exactly zero. The factorization has been completed, but the block diagonal matrix \( D \) is exactly singular, so the solution and error bounds could not be computed; \( rcond = 0 \) is returned.

If \( info = i \), and \( i = n + 1 \), then \( D \) is nonsingular, but \( rcond \) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \( rcond \) would suggest.

**See Also**

Matrix Storage Schemes

`?sysvxx`

*Uses extra precise iterative refinement to compute the solution to the system of linear equations with a symmetric indefinite coefficient matrix \( A \) applying the diagonal pivoting factorization.*

**Syntax**

```c
lapack_int LAPACKE_ssysvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, double* b, lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );
```

```c
lapack_int LAPACKE_dsysvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, double* b, lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

```c
lapack_int LAPACKE_csysvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );
```

```c
lapack_int LAPACKE_zsysvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

**Include Files**

- mkl.h

**Description**

The routine uses the *diagonal pivoting* factorization to compute the solution to a real or complex system of linear equations \( A^tX = B \), where \( A \) is an \( n \)-by-\( n \) real symmetric/Hermitian matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error \( O(\text{eps}) \), where \( \text{eps} \) is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.
The routine accepts user-provided factorizations and equilibration factors; see definitions of the fact and equed options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\varepsilon)$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine ?sysvxx performs the following steps:

1. If fact = 'E', scaling factors are computed to equilibrate the system:
   \[ \text{diag}(s) \cdot A \cdot \text{diag}(s) \cdot \text{inv}(\text{diag}(s)) \cdot X = \text{diag}(s) \cdot B \]
   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(s) \cdot A \cdot \text{diag}(s)$ and $B$ by $\text{diag}(s) \cdot B$.

2. If fact = 'N' or 'E', the LU decomposition is used to factor the matrix $A$ (after equilibration if fact = 'E') as
   \[
   \begin{align*}
   A &= U \cdot D \cdot U^T, & \text{if } \text{uplo} = 'U', \\
   A &= L \cdot D \cdot L^T, & \text{if } \text{uplo} = 'L',
   \end{align*}
   \]
   where $U$ or $L$ is a product of permutation and unit upper (lower) triangular matrices, and $D$ is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

3. If some $D(i,i)=0$, so that $D$ is exactly singular, the routine returns with info = i. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$ (see the rcond parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for $X$ and compute error bounds.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. By default, unless params[0] is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(r)$ so that it solves the original system before equilibration.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **fact**: Must be 'F', 'N', or 'E'. Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.

  - If fact = 'F', on entry, $af$ and $ipiv$ contain the factored form of $A$.
  - If equed is not 'N', the matrix $A$ has been equilibrated with scaling factors given by $s$. Parameters $a$, $af$, and $ipiv$ are not modified.
  - If fact = 'N', the matrix $A$ will be copied to $af$ and factored.
  - If fact = 'E', the matrix $A$ will be equilibrated, if necessary, copied to $af$ and factored.

- **uplo**: Must be 'U' or 'L'. Indicates whether the upper or lower triangular part of $A$ is stored:

  - If uplo = 'U', the upper triangle of $A$ is stored.
  - If uplo = 'L', the lower triangle of $A$ is stored.

- **n**: The number of linear equations; the order of the matrix $A$; $n \geq 0$. 

---

Intel® Math Kernel Library Developer Reference

692
The number of right-hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$.

Arrays: $a(\text{size max}(1, \text{lda}*n))$, $af(\text{size max}(1, \text{ldaf}*n))$, $b$, $\text{size max}(\text{ldb}*nrhs)$ for column major layout and $\text{max}(\text{ldb}*n)$ for row major layout.

The array $a$ contains the symmetric matrix $A$ as specified by $\text{uplo}$. If $\text{uplo} = 'U'$, the leading $n$-by-$n$ upper triangular part of $a$ contains the upper triangular part of the matrix $A$ and the strictly lower triangular part of $a$ is not referenced. If $\text{uplo} = 'L'$, the leading $n$-by-$n$ lower triangular part of $a$ contains the lower triangular part of the matrix $A$ and the strictly upper triangular part of $a$ is not referenced.

The array $af$ is an input argument if $\text{fact} = 'F'$. It contains the block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ and $L$ from the factorization $A = U*D*U^T$ or $A = L*D*L^T$ as computed by ?sytrf.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

The leading dimension of the array $a$; $\text{lda} \geq \text{max}(1, n)$.

The leading dimension of the array $af$; $\text{ldaf} \geq \text{max}(1, n)$.

Array, size at least $\text{max}(1, n)$. The array $ipiv$ is an input argument if $\text{fact} = 'F'$. It contains details of the interchanges and the block structure of $D$ as determined by ?sytrf. If $ipiv[k-1] > 0$, rows and columns $k$ and $ipiv[k-1]$ were interchanged and $D(k,k)$ is a 1-by-1 diagonal block.

If $\text{uplo} = 'U$' and $ipiv[i] = ipiv[i - 1] = m < 0$, $D$ has a 2-by-2 diagonal block in rows and columns $i$ and $i + 1$, and the $i$-th row and column of $A$ were interchanged with the $m$-th row and column.

If $\text{uplo} = 'L$' and $ipiv[i] = ipiv[i - 1] = m < 0$, $D$ has a 2-by-2 diagonal block in rows and columns $i$ and $i + 1$, and the $(i + 1)$-st row and column of $A$ were interchanged with the $m$-th row and column.

Must be 'N' or 'Y'.

equed is an input argument if $\text{fact} = 'F'$. It specifies the form of equilibration that was done:

If $\text{equed} = 'N'$, no equilibration was done (always true if $\text{fact} = 'N'$).

if $\text{equed} = 'Y'$, both row and column equilibration was done, that is, $A$ has been replaced by $\text{diag}(s)*A*\text{diag}(s)$.

Array, size $(n)$. The array $s$ contains the scale factors for $A$. If $\text{equed} = 'Y'$, $A$ is multiplied on the left and right by $\text{diag}(s)$.

This array is an input argument if $\text{fact} = 'F'$ only; otherwise it is an output argument.

If $\text{fact} = 'F'$ and $\text{equed} = 'Y'$, each element of $s$ must be positive.
Each element of $s$ should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

$ldb$

The leading dimension of the array $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

$ldx$

The leading dimension of the output array $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

$n_{err\_bnds}$

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See $err\_bnds\_norm$ and $err\_bnds\_comp$ descriptions in the Output Arguments section below.

$nparams$

Specifies the number of parameters set in $params$. If $\leq 0$, the $params$ array is never referenced and default values are used.

$params$

Array, size $\max(1, nparams)$. Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to $nparams$ are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass $nparams = 0$, which prevents the source code from accessing the $params$ argument.

$params[0]$: Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).

$=0.0$ No refinement is performed and no error bounds are computed.

$=1.0$ Use the extra-precise refinement algorithm.

(Other values are reserved for future use.)

$params[1]$: Maximum number of residual computations allowed for refinement.

Default 10.0

Aggressive Set to 100.0 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in $err\_bnds\_norm$ and $err\_bnds\_comp$ may no longer be trustworthy.

$params[2]$: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).
Output Parameters

$x$

Array, size $\text{max}(1, \ ldx*\text{nrhs})$ for column major layout and $\text{max}(1, \ ldx*n)$ for row major layout.

If $\text{info} = 0$, the array $x$ contains the solution $n$-by-$\text{nrhs}$ matrix $X$ to the original system of equations. Note that $A$ and $B$ are modified on exit if $\text{equed} \neq 'N'$, and the solution to the equilibrated system is:

$$\text{inv(diag}(s)\text{)*}X.$$

If $\text{fact} = 'E'$ and $\text{equed} = 'Y'$, overwritten by $\text{diag}(s)*A*\text{diag}(s)$.

$a$

If $\text{fact} = 'N'$, $af$ is an output argument and on exit returns the block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ or $L$ from the factorization $A = U*D*U^T$ or $A = L*D*L^T$.

$b$

If $\text{equed} = 'N'$, $B$ is not modified.

If $\text{equed} = 'Y'$, $B$ is overwritten by $\text{diag}(s)*B$.

$s$

This array is an output argument if $\text{fact} \neq 'F'$'. Each element of this array is a power of the radix. See the description of $s$ in Input Arguments section.

$rcond$

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix $A$ after equilibration (if done). If $rcond$ is less than the machine precision, in particular, if $rcond = 0$, the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

$rpvgrw$

Contains the reciprocal pivot growth factor:

$$\|A\|/\|U\|$$

If this is much less than 1, the stability of the $LU$ factorization of the (equilibrated) matrix $A$ could be poor. This also means that the solution $X$, estimated condition numbers, and error bounds could be unreliable. If factorization fails with $0 < \text{info} \leq n$, this parameter contains the reciprocal pivot growth factor for the leading $\text{info}$ columns of $A$.

$berr$

Array, size at least $\text{max}(1, \ \text{nrhs})$. Contains the componentwise relative backward error for each solution vector $x_j$, that is, the smallest relative change in any element of $A$ or $B$ that makes $x_j$ an exact solution.

$\text{err_bnds_norm}$

Array of size $\text{nrhs}*\text{n_err_bnds}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the $i$-th solution vector

$$\max_j \left|X_{true}^{ji} - X_{ji}\right| /
\max_j \left|X_{ji}\right|$$

The array is indexed by the type of error information as described below. Up to three pieces of information are returned.
**err=1**  
"Trust/don’t trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors.

**err=2**  
"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors. This error bound should only be trusted if the previous boolean is true.

**err=3**  
Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix \( Z \) are:

\[
\|Z\|_\infty \cdot \|Z^{-1}\|_\infty
\]

Let \( z = s \times a \), where \( s \) scales each row by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( err \) is stored in \( \text{err_bnds_norm}[(\text{err}-1) \times \text{nrhs} + i - 1] \).

Array of size \( \text{nrhs} \times \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the \( i \)-th solution vector:

\[
\max_j \frac{|X_{true_{ij}} - X_{ij}|}{|X_{ij}|}
\]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested (\( \text{params}[2] = 0.0 \)), then \( \text{err_bnds_comp} \) is not accessed.

**err=1**  
"Trust/don’t trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \times \text{slamch}(\varepsilon) \) for single precision flavors and \( \sqrt{n} \times \text{dlamch}(\varepsilon) \) for double precision flavors.
err=2

"Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors. This error bound should only be trusted if the previous boolean is true.

err=3

Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} \times \text{slamch}(\varepsilon)$ for single precision flavors and $\sqrt{n} \times \text{dlamch}(\varepsilon)$ for double precision flavors to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\| \cdot \|Z^{-1}\|$$

Let $z = s \times (a \times \text{diag}(x))$, where $x$ is the solution for the current right-hand side and $s$ scales each row of $a \times \text{diag}(x)$ by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error err is stored in $\text{err_bnds_comp}[(\text{err}-1) \times \text{nrhs} + i - 1]$.

ipiv

If fact = 'N', ipiv is an output argument and on exit contains details of the interchanges and the block structure $D_i$ as determined by ssytrf for single precision flavors and dsytrf for double precision flavors.

equed

If fact≠'F', then equed is an output argument. It specifies the form of equilibration that was done (see the description of equed in Input Arguments section).

params

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

Return Values

This function returns a value info.

If info = 0, the execution is successful. The solution to every right-hand side is guaranteed.

If info = -i, parameter i had an illegal value.

If 0 < info<n: $U_{info,info}$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution and error bounds could not be computed; rcond = 0 is returned.

If info = n+j: The solution corresponding to the j-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides $k$ with $k > j$ may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested params[2] = 0.0, then the j-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest $j$ such that for column major layout $\text{err_bnds_norm}[j - 1] = 0.0$ or $\text{err_bnds_comp}[j - 1] = 0.0$; or for row major
The routine solves for \( X \) the complex system of linear equations \( A^*X = B \), where \( A \) is an \( n \)-by-\( n \) symmetric matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

The diagonal pivoting method is used to factor \( A \) as \( U*D*U^H \) or \( L*D*L^H \), where \( U \) (or \( L \)) is a product of permutation and unit upper (lower) triangular matrices, and \( D \) is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of \( A \) is then used to solve the system of equations \( A^*X = B \).

### Input Parameters

- **\( \text{matrix_layout} \)**
  - Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).
  - Must be 'U' or 'L'.

- **\( \text{uplo} \)**
  - Indicates whether the upper or lower triangular part of \( A \) is stored and how \( A \) is factored:
    - If \( \text{uplo} = 'U' \), the array \( a \) stores the upper triangular part of the matrix \( A \), and \( A \) is factored as \( U*D*U^H \).
    - If \( \text{uplo} = 'L' \), the array \( a \) stores the lower triangular part of the matrix \( A \), and \( A \) is factored as \( L*D*L^H \).

- **\( n \)**
  - The order of matrix \( A \); \( n \geq 0 \).

- **\( \text{nrhs} \)**
  - The number of right-hand sides, the number of columns in \( B \); \( \text{nrhs} \geq 0 \).
Arrays: $a$ (size max(1, $lda \cdot n$)), $b$ of size max(1, $ldb \cdot nrhs$) for column major layout and max(1, $ldb \cdot n$) for row major layout. The array $a$ contains the upper or the lower triangular part of the Hermitian matrix $A$ (see $uplo$).

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

The leading dimension of $a$; $lda \geq \max(1, n)$.

The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

Output Parameters

$a$

If $info = 0$, $a$ is overwritten by the block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$) from the factorization of $A$ as computed by $\text{?hetrf}$.

$b$

If $info = 0$, $b$ is overwritten by the solution matrix $X$.

$ipiv$

Array, size at least max(1, n). Contains details of the interchanges and the block structure of $D$, as determined by $\text{?hetrf}$.

If $ipiv[i-1] = k > 0$, then $d_{ii}$ is a 1-by-1 diagonal block, and the $i$-th row and column of $A$ was interchanged with the $k$-th row and column.

If $uplo = 'U$' and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $(i)$-th row and column of $A$ was interchanged with the $m$-th row and column.

If $uplo = 'L$' and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $(i+1)$-th row and column of $A$ was interchanged with the $m$-th row and column.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, parameter $i$ had an illegal value.

If $info = i$, $d_{ii}$ is 0. The factorization has been completed, but $D$ is exactly singular, so the solution could not be computed.

See Also

Matrix Storage Schemes

$\text{hesv_aa}$

Computes the solution to system of linear equations for HE matrices.

LAPACK DECL lapack_int LAPACKE_chesv_aa (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb );
LAPACK DECL lapack_int LAPACKE_chesv_aa_work (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * work, lapack_int lwork);

Description

?hesv_aa computes the solution to a complex system of linear equations \( A \times X = B \), where \( A \) is an \( n \)-by-\( n \) Hermitian matrix and \( X \) and \( B \) are \( n \)-by-\( nrhs \) matrices. Aasen’s algorithm is used to factor \( A \) as

\[
A = U \times T \times U^H \text{ if } uplo = 'U', \text{ or } \\
A = L \times T \times L^H \text{ if } uplo = 'L',
\]

where \( U \) (or \( L \)) is a product of permutation and unit upper (lower) triangular matrices, and \( T \) is Hermitian and tridiagonal. The factored form of \( A \) is then used to solve the system of equations \( A \times X = B \).

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_layout</td>
<td>Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).</td>
</tr>
<tr>
<td>uplo</td>
<td>If ( uplo = 'U' ): The upper triangle of ( A ) is stored. If ( uplo = 'L' ): the lower triangle of ( A ) is stored.</td>
</tr>
<tr>
<td>n</td>
<td>The number of linear equations or the order of the matrix ( A ). ( n \geq 0 ).</td>
</tr>
<tr>
<td>nrhs</td>
<td>The number of right hand sides or the number of columns of the matrix ( B ). ( nrhs \geq 0 ).</td>
</tr>
<tr>
<td>a</td>
<td>Array of size ( lda \times n ). On entry, the Hermitian matrix ( A ). If ( uplo = 'U' ), the leading ( n )-by-( n ) upper triangular part of ( a ) contains the upper triangular part of the matrix ( A ), and the strictly lower triangular part of ( a ) is not referenced. If ( uplo = 'L' ), the leading ( n )-by-( n ) lower triangular part of ( a ) contains the lower triangular part of the matrix ( A ), and the strictly upper triangular part of ( a ) is not referenced.</td>
</tr>
<tr>
<td>lda</td>
<td>The leading dimension of the array ( a ). ( lda \geq \max(1,n) ).</td>
</tr>
<tr>
<td>b</td>
<td>Array of size ( ldb \times nrhs ). On entry, the ( n )-by-( nrhs ) right hand side matrix ( B ).</td>
</tr>
<tr>
<td>ldb</td>
<td>The leading dimension of the array ( b ). ( ldb \geq \max(1,n) ).</td>
</tr>
<tr>
<td>lwork</td>
<td>The length of ( work ). ( lwork \geq \max(1, 2*\text{n}, 3*\text{n}-2) ), and for best performance ( lwork \geq \max(1,\text{n}*\text{nb}) ), where ( \text{nb} ) is the optimal blocksize for (?hetrf). If ( lwork &lt; n ), TRS is done with Level BLAS 2. If ( lwork \geq n ), TRS is done with Level BLAS 3. If ( lwork = -1 ), then a workspace query is assumed; the routine only calculates the optimal size of the ( work ) array, returns this value as the first entry of the ( work ) array, and no error message related to ( lwork ) is issued by \text{xerbla}.</td>
</tr>
</tbody>
</table>
Output Parameters

\( a \)
On exit, if \( info = 0 \), the tridiagonal matrix \( T \) and the multipliers used to obtain the factor \( U \) or \( L \) from the factorization \( A = U^*T^*U^H \) or \( A = L^*T^*L^H \) as computed by ?hetrf_aa.

\( ipiv \)
Array of size \( (n) \) On exit, it contains the details of the interchanges: row and column \( k \) of \( A \) were interchanged with the row and column \( ipiv[k] \).

\( b \)
On exit, if \( info = 0 \), the \( n \)-by-\( nrhs \) solution matrix \( X \).

\( work \)
Array of size \( (\max(1, lwork)) \). On exit, if \( info = 0 \), \( work[0] \) returns the optimal \( lwork \).

Return Values

This function returns a value \( info \).

If \( info = 0 \): successful exit.

If \( info < 0 \): if \( info = -i \), the \( i \)-th argument had an illegal value.

If \( info > 0 \): if \( info = i \), \( D_{i,i} \) is exactly zero. The factorization has been completed, but the block diagonal matrix \( D \) is exactly singular, so the solution could not be computed.

?hesv_rk
?hesv_rk computes the solution to a system of linear equations \( A \times X = B \) for Hermitian matrices.

```
lapack_int LAPACK_chesv_rk (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * A, lapack_int lda, lapack_complex_float * e, lapack_int * ipiv, lapack_complex_float * B, lapack_int ldb);
```

Description

?hesv_rk computes the solution to a complex system of linear equations \( A \times X = B \), where \( A \) is an \( n \)-by-\( n \) Hermitian matrix and \( X \) and \( B \) are \( n \)-by-\( nrhs \) matrices.

The bounded Bunch-Kaufman (rook) diagonal pivoting method is used to factor \( A \) as \( A = P \times U^*D^*(U^H)(P^T) \), if \( uplo = 'U' \), or \( A = P \times L^*D^*(L^H)(P^T) \), if \( uplo = 'L' \), where \( U \) (or \( L \)) is unit upper (or lower) triangular matrix, \( U^H \) (or \( L^H \)) is the conjugate of \( U \) (or \( L \)), \( P \) is a permutation matrix, \( P^T \) is the transpose of \( P \), and \( D \) is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

?hetrf_rk is called to compute the factorization of a complex Hermitian matrix. The factored form of \( A \) is then used to solve the system of equations \( A \times X = B \) by calling BLAS3 routine ?hetrs_3.

Input Parameters

\( matrix_layout \)
Specifies whether matrix storage layout is row major (\( \text{LAPACK\_ROW\_MAJOR} \)) or column major (\( \text{LAPACK\_COL\_MAJOR} \)).

\( uplo \)
Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is stored:
- \( '=' \) \( 'U' \): The upper triangle of \( A \) is stored.
- \( '=' \) \( 'L' \): The lower triangle of \( A \) is stored.
The number of linear equations; that is, the order of the matrix A. \( n \geq 0 \).

The number of right-hand sides; that is, the number of columns of the matrix B. \( nrhs \geq 0 \).

Array of size \( \max(1, lda* n) \). On entry, the Hermitian matrix A. If \( uplo = 'U' \): the leading \( n \)-by-\( n \) upper triangular part of A contains the upper triangular part of the matrix A, and the strictly lower triangular part of A is not referenced. If \( uplo = 'L' \): the leading \( n \)-by-\( n \) lower triangular part of A contains the lower triangular part of the matrix A, and the strictly upper triangular part of A is not referenced.

The leading dimension of the array A.

On entry, the \( n \)-by-\( nrhs \) right-hand side matrix B.

The size of B is \( \max(1, ldb*nrhs) \) for column-major layout and \( \max(1, ldb*n) \) for row-major layout.

The leading dimension of the array B. \( ldb \geq \max(1, n) \) for column-major layout and \( ldb \geq nrhs \) for row-major layout.

Output Parameters

A

On exit, if \( info = 0 \), diagonal of the block diagonal matrix D and factors U or L as computed by \( ?hetrf_rk \):

- Only diagonal elements of the Hermitian block diagonal matrix D on the diagonal of A; that is, \( D(k,k) = A(k,k) \); (superdiagonal (or subdiagonal) elements of D are stored on exit in array e).

  -and-

- If \( uplo = 'U' \), factor U in the superdiagonal part of A. If \( uplo = 'L' \), factor L in the subdiagonal part of A.

For more information, see the description of the \( ?hetrf_rk \) routine.

Array of size \( n \). On exit, contains the output computed by the factorization routine \( ?hetrf_rk \); that is, the superdiagonal (or subdiagonal) elements of the Hermitian block diagonal matrix D with 1-by-1 or 2-by-2 diagonal blocks:

- If \( uplo = 'U' \), \( e(i) = D(i-1,i), i=2:N, e(1) \) is set to 0.
- If \( uplo = 'L' \), \( e(i) = D(i+1,i), i=1:N-1, e(n) \) is set to 0.

**NOTE** For a 1-by-1 diagonal block \( D(k) \), where \( 1 \leq k \leq n \), the element \( e(k) \) is set to 0 in both the \( uplo = 'U' \) and \( uplo = 'L' \) cases.

For more information, see the description of the \( ?hetrf_rk \) routine.

ipiv

Array of size \( n \). Details of the interchanges and the block structure of D, as determined by \( ?hetrf_rk \).

B

On exit, if \( info = 0 \), the \( n \)-by-\( nrhs \) solution matrix X.
**Return Values**

This function returns a value \( info \).

- \( = 0 \): Successful exit.
- \(< 0\): If \( info = -k \), the \( k \)th argument had an illegal value.
- \( > 0\): If \( info = k \), the matrix \( A \) is singular. If \( uplo = 'U' \), column \( k \) in the upper triangular part of \( A \) contains all zeros. If \( uplo = 'L' \), column \( k \) in the lower triangular part of \( A \) contains all zeros. Therefore \( D(k,k) \) is exactly zero, and superdiagonal elements of column \( k \) of \( U \) (or subdiagonal elements of column \( k \) of \( L \)) are all zeros. The factorization has been completed, but the block diagonal matrix \( D \) is exactly singular, and division by zero will occur if it is used to solve a system of equations.

**hesvx**

*Uses the diagonal pivoting factorization to compute the solution to the complex system of linear equations with a Hermitian coefficient matrix \( A \), and provides error bounds on the solution.*

**Syntax**

```c
lapack_int LAPACKE_chesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* a, lapack_int lda, lapack_complex_float* af,
lapack_int ldaf, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zhesvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* a, lapack_int lda, lapack_complex_double* af,
lapack_int ldaf, lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

**Include Files**

- mkl.h

**Description**

The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations \( A^*X = B \), where \( A \) is an \( n \)-by-\( n \) Hermitian matrix, the columns of matrix \( B \) are individual right-hand sides, and the columns of \( X \) are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine \( ?hesvx \) performs the following steps:

1. If \( fact = 'N' \), the diagonal pivoting method is used to factor the matrix \( A \). The form of the factorization is \( A = U^*D^*U \) or \( A = L^*D^*L \), where \( U \) (or \( L \)) is a product of permutation and unit upper (lower) triangular matrices, and \( D \) is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
2. If some \( d_{i,i} = 0 \), so that \( D \) is exactly singular, then the routine returns with \( info = i \). Otherwise, the factored form of \( A \) is used to estimate the condition number of the matrix \( A \). If the reciprocal of the condition number is less than machine precision, \( info = n+1 \) is returned as a warning, but the routine still goes on to solve for \( X \) and compute error bounds as described below.
3. The system of equations is solved for \( X \) using the factored form of \( A \).
4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.
Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

fact

Must be 'F' or 'N'.

Specifies whether or not the factored form of the matrix A has been supplied on entry.

If fact = 'F': on entry, af and ipiv contain the factored form of A. Arrays a, af, and ipiv are not modified.

If fact = 'N', the matrix A is copied to af and factored.

uplo

Must be 'U' or 'L'.

Indicates whether the upper or lower triangular part of A is stored and how A is factored:

If uplo = 'U', the array a stores the upper triangular part of the Hermitian matrix A, and A is factored as $U^*D*U^H$.

If uplo = 'L', the array a stores the lower triangular part of the Hermitian matrix A; A is factored as $L^*D*L^H$.

n

The order of matrix A; $n \geq 0$.

nrhs

The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

a, af, b

Arrays: $a$ (size max(1, lda*n)), $af$ (size max(1, ldaf*n)), $b$ of size max(1, ldb*nrhs) for column major layout and max(1, ldb*n) for row major layout.

The array a contains the upper or the lower triangular part of the Hermitian matrix A (see uplo).

The array af is an input argument if fact = 'F'. It contains he block diagonal matrix D and the multipliers used to obtain the factor $U$ or $L$ from the factorization $A = U^*D*U^H$ or $A = L^*D*L^H$ as computed by ?hetrf.

The array b contains the matrix B whose columns are the right-hand sides for the systems of equations.

lda

The leading dimension of $a$; $lda \geq \text{max}(1, n)$.

ldaf

The leading dimension of $af$; $ldaf \geq \text{max}(1, n)$.

ldb

The leading dimension of $b$; $ldb \geq \text{max}(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ipiv

Array, size at least max(1, n). The array ipiv is an input argument if fact = 'F'. It contains details of the interchanges and the block structure of $D$, as determined by ?hetrf.

If $ipiv[i-1] = k > 0$, then $d_{ii}$ is a 1-by-1 diagonal block, and the i-th row and column of A was interchanged with the k-th row and column.
If uplo = 'U' and ipiv[i] = ipiv[i-1] = -m < 0, then D has a 2-by-2 block in rows/columns i and i+1, and (i)-th row and column of A was interchanged with the m-th row and column.

If uplo = 'L' and ipiv[i] = ipiv[i-1] = -m < 0, then D has a 2-by-2 block in rows/columns i and i+1, and (i+1)-th row and column of A was interchanged with the m-th row and column.

ldx
The leading dimension of the output array x; ldx≥ max(1, n) for column major layout and ldx≥nrhs for row major layout.

Output Parameters

x
Array, size max(1, ldx*nrhs) for column major layout and max(1, ldx*n) for row major layout.
If info = 0 or info = n+1, the array x contains the solution matrix X to the system of equations.

af, ipiv
These arrays are output arguments if fact = 'N'. See the description of af, ipiv in Input Arguments section.

rcond
An estimate of the reciprocal condition number of the matrix A. If rcond is less than the machine precision (in particular, if rcond = 0), the matrix is singular to working precision. This condition is indicated by a return code of info > 0.

ferr
Array, size at least max(1, nrhs). Contains the estimated forward error bound for each solution vector x_j (the j-th column of the solution matrix X). If xtrue is the true solution corresponding to x_j, ferr[j-1] is an estimated upper bound for the magnitude of the largest element in (x_j - xtrue) divided by the magnitude of the largest element in x_j. The estimate is as reliable as the estimate for rcon, and is almost always a slight overestimate of the true error.

berr
Array, size at least max(1, nrhs). Contains the component-wise relative backward error for each solution vector x_j, that is, the smallest relative change in any element of A or B that makes x_j an exact solution.

Return Values

This function returns a value info.
If info = 0, the execution is successful.
If info = -i, parameter i had an illegal value.
If info = i, and i≤n, then d_ii is exactly zero. The factorization has been completed, but the block diagonal matrix D is exactly singular, so the solution and error bounds could not be computed; rcond = 0 is returned.
If info = i, and i = n + 1, then D is nonsingular, but rcond is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of rcond would suggest.

See Also
Matrix Storage Schemes
**hesvxx**

Uses extra precise iterative refinement to compute the solution to the system of linear equations with a Hermitian indefinite coefficient matrix $A$ applying the diagonal pivoting factorization.

**Syntax**

```c
lapack_int LAPACKE_chesvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* af, lapack_int ldaf, lapack_int* ipiv, char* equed, float* s, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* rpvgrw, float* berr, lapack_int n_err_bnds, float* err_bnds_norm, float* err_bnds_comp, lapack_int nparams, const float* params );
```

```c
lapack_int LAPACKE_zhesvxx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* af, lapack_int ldaf, lapack_int* ipiv, char* equed, double* s, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* x, lapack_int ldx, double* rcond, double* rpvgrw, double* berr, lapack_int n_err_bnds, double* err_bnds_norm, double* err_bnds_comp, lapack_int nparams, const double* params );
```

**Include Files**

- mkl.h

**Description**

The routine uses the diagonal pivoting factorization to compute the solution to a complex/double complex system of linear equations $A^*X = B$, where $A$ is an $n$-by-$n$ Hermitian matrix, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Both normwise and maximum componentwise error bounds are also provided on request. The routine returns a solution with a small guaranteed error ($O(\varepsilon)$, where $\varepsilon$ is the working machine precision) unless the matrix is very ill-conditioned, in which case a warning is returned. Relevant condition numbers are also calculated and returned.

The routine accepts user-provided factorizations and equilibration factors; see definitions of the fact and equed options. Solving with refinement and using a factorization from a previous call of the routine also produces a solution with $O(\varepsilon)$ errors or warnings but that may not be true for general user-provided factorizations and equilibration factors if they differ from what the routine would itself produce.

The routine **hesvxx** performs the following steps:

1. If **fact** = 'E', scaling factors are computed to equilibrate the system:
   
   $\text{diag}(s)^*A^{*}\text{diag}(s)^*\text{inv}(\text{diag}(s))^*X = \text{diag}(s)^*B$
   
   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(s)^*A^{*}\text{diag}(s)$ and $B$ by $\text{diag}(s)^*B$.

2. If **fact** = 'N' or 'E', the LU decomposition is used to factor the matrix $A$ (after equilibration if **fact** = 'E') as
   
   $A = U^{*}D^{*}U^{T}$, if uplo = 'U',
   
   or $A = L^{*}D^{*}L^{T}$, if uplo = 'L',
   
   where $U$ or $L$ is a product of permutation and unit upper (lower) triangular matrices, and $D$ is a symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.
3. If some $D(i,i)=0$, so that $D$ is exactly singular, the routine returns with $info = i$. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$ (see the $rcond$ parameter). If the reciprocal of the condition number is less than machine precision, the routine still goes on to solve for $X$ and compute error bounds.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. By default, unless $params[0]$ is set to zero, the routine applies iterative refinement to get a small error and error bounds. Refinement calculates the residual to at least twice the working precision.

6. If equilibration was used, the matrix $X$ is premultiplied by $diag(r)$ so that it solves the original system before equilibration.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **fact**: Must be 'F', 'N', or 'E'.

  Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.

  If fact = 'F', on entry, $af$ and $ipiv$ contain the factored form of $A$. If $equed$ is not 'N', the matrix $A$ has been equilibrated with scaling factors given by $s$. Parameters $a$, $af$, and $ipiv$ are not modified.

  If fact = 'N', the matrix $A$ will be copied to $af$ and factored.

  If fact = 'E', the matrix $A$ will be equilibrated, if necessary, copied to $af$ and factored.

- **uplo**: Must be 'U' or 'L'.

  Indicates whether the upper or lower triangular part of $A$ is stored:

  If uplo = 'U', the upper triangle of $A$ is stored.

  If uplo = 'L', the lower triangle of $A$ is stored.

- **n**: The number of linear equations; the order of the matrix $A$; $n \geq 0$.

- **nrhs**: The number of right-hand sides; the number of columns of the matrices $B$ and $X$; $nrhs \geq 0$.

- **a, af, b**: Arrays: $a$ (size max($lda*n$)), $af$ (size max($ldaf*n$)), $b$ (size max($ldb*nrhs$) for column major layout and max($ldb*n$) for row major layout).

  The array $a$ contains the Hermitian matrix $A$ as specified by uplo. If uplo = 'U', the leading $n$-by-$n$ upper triangular part of $a$ contains the upper triangular part of the matrix $A$ and the strictly lower triangular part of $a$ is not referenced. If uplo = 'L', the leading $n$-by-$n$ lower triangular part of $a$ contains the lower triangular part of the matrix $A$ and the strictly upper triangular part of $a$ is not referenced.

  The array $af$ is an input argument if fact = 'F'. It contains the block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ and $L$ from the factorization $A = U^T*D*U$ or $A = L*D*L^T$ as computed by ?hetrf.
The array \( b \) contains the matrix \( B \) whose columns are the right-hand sides for the systems of equations.

\[ \text{lda} \]

The leading dimension of the array \( a \); \( \text{lda} \geq \max(1, n) \).

\[ \text{ldaf} \]

The leading dimension of the array \( af \); \( \text{ldaf} \geq \max(1, n) \).

\[ \text{ipiv} \]

Array, size at least \( \max(1, n) \). The array \( \text{ipiv} \) is an input argument if \( \text{fact} = 'F' \). It contains details of the interchanges and the block structure of \( D \) as determined by \( \text{?sytrf} \).

If \( \text{ipiv}[k-1] > 0 \), rows and columns \( k \) and \( \text{ipiv}[k-1] \) were interchanged and \( D_{k,k} \) is a 1-by-1 diagonal block.

If \( \text{uplo} = 'U' \) and \( \text{ipiv}[i] = \text{ipiv}[i - 1] = m < 0 \), \( D \) has a 2-by-2 diagonal block in rows and columns \( i \) and \( i + 1 \), and the \( i \)-th row and column of \( A \) were interchanged with the \( m \)-th row and column.

If \( \text{uplo} = 'L' \) and \( \text{ipiv}[i] = \text{ipiv}[i - 1] = m < 0 \), \( D \) has a 2-by-2 diagonal block in rows and columns \( i \) and \( i + 1 \), and the \((i + 1)\)-st row and column of \( A \) were interchanged with the \( m \)-th row and column.

\[ \text{equed} \]

Must be \( 'N' \) or \( 'Y' \).

\( \text{equed} \) is an input argument if \( \text{fact} = 'F' \). It specifies the form of equilibration that was done:

If \( \text{equed} = 'N' \), no equilibration was done (always true if \( \text{fact} = 'N' \)).

If \( \text{equed} = 'Y' \), both row and column equilibration was done, that is, \( A \) has been replaced by \( \text{diag}(s) \cdot A \cdot \text{diag}(s) \).

\[ s \]

Array, size \( (n) \). The array \( s \) contains the scale factors for \( A \). If \( \text{equed} = 'Y' \), \( A \) is multiplied on the left and right by \( \text{diag}(s) \).

This array is an input argument if \( \text{fact} = 'F' \) only; otherwise it is an output argument.

If \( \text{fact} = 'F' \) and \( \text{equed} = 'Y' \), each element of \( s \) must be positive.

Each element of \( s \) should be a power of the radix to ensure a reliable solution and error estimates. Scaling by powers of the radix does not cause rounding errors unless the result underflows or overflows. Rounding errors during scaling lead to refining with a matrix that is not equivalent to the input matrix, producing error estimates that may not be reliable.

\[ \text{ldb} \]

The leading dimension of the array \( b \); \( \text{ldb} \geq \max(1, n) \) for column major layout and \( \text{ldb} \geq \text{nrhs} \) for row major layout.

\[ \text{ldx} \]

The leading dimension of the output array \( x \); \( \text{ldx} \geq \max(1, n) \) for column major layout and \( \text{ldx} \geq \text{nrhs} \) for row major layout.

\[ n\_err\_bnds \]

Number of error bounds to return for each right hand side and each type (normwise or componentwise). See \( \text{err_bnds_norm} \) and \( \text{err_bnds_comp} \) descriptions in the Output Arguments section below.
Specifies the number of parameters set in `params`. If ≤ 0, the `params` array is never referenced and default values are used.

`params` Array, size max(1, `nparams`). Specifies algorithm parameters. If an entry is less than 0.0, that entry is filled with the default value used for that parameter. Only positions up to `nparams` are accessed; defaults are used for higher-numbered parameters. If defaults are acceptable, you can pass `nparams = 0`, which prevents the source code from accessing the `params` argument.

- `params[0]`: Whether to perform iterative refinement or not. Default: 1.0 (for single precision flavors), 1.0D+0 (for double precision flavors).
  - =0.0 No refinement is performed and no error bounds are computed.
  - =1.0 Use the extra-precise refinement algorithm.
  (Other values are reserved for future use.)
- `params[1]`: Maximum number of residual computations allowed for refinement.
  - Default: 10
  - Aggressive: Set to 100 to permit convergence using approximate factorizations or factorizations other than LU. If the factorization uses a technique other than Gaussian elimination, the guarantees in `err_bnds_norm` and `err_bnds_comp` may no longer be trustworthy.
- `params[2]`: Flag determining if the code will attempt to find a solution with a small componentwise relative error in the double-precision algorithm. Positive is true, 0.0 is false. Default: 1.0 (attempt componentwise convergence).

### Output Parameters

- `x`: Array, size max(1, `ldx*nrhs`) for column major layout and max(1, `ldx*n`) for row major layout.
  - If `info = 0`, the array `x` contains the solution `n`-by-`nrhs` matrix `X` to the original system of equations. Note that `A` and `B` are modified on exit if `equed='N'`, and the solution to the equilibrated system is: `inv(diag(s))*X`.
  - If `fact = 'E'` and `equed = 'Y'`, overwritten by `diag(s)*A*diag(s)`.
- `af`:
  - If `fact = 'N'`, `af` is an output argument and on exit returns the block diagonal matrix `D` and the multipliers used to obtain the factor `U` or `L` from the factorization `A = U*D*U^T` or `A = L*D*L^T`.
  - If `equed = 'N'`, `B` is not modified.
If \texttt{equed} = 'Y', \( B \) is overwritten by \( \text{diag}(s) \cdot B \).

This array is an output argument if \texttt{fact} = 'F'. Each element of this array is a power of the radix. See the description of \( s \) in Input Arguments section.

\( s \)

Reciprocal scaled condition number. An estimate of the reciprocal Skeel condition number of the matrix \( A \) after equilibration (if done). If \( rcond \) is less than the machine precision, in particular, if \( rcond = 0 \), the matrix is singular to working precision. Note that the error may still be small even if this number is very small and the matrix appears ill-conditioned.

\( rcond \)

Contains the reciprocal pivot growth factor:

\[ \frac{\|A\|}{\|U\|} \]

If this is much less than 1, the stability of the \( LU \) factorization of the (equilibrated) matrix \( A \) could be poor. This also means that the solution \( X \), estimated condition numbers, and error bounds could be unreliable. If factorization fails with \( 0 < \text{info} \leq n \), this parameter contains the reciprocal pivot growth factor for the leading \( \text{info} \) columns of \( A \).

\( rpvgrw \)

Array, size at least \( \max(1, \text{nrhs}) \). Contains the component-wise relative backward error for each solution vector \( x_j \), that is, the smallest relative change in any element of \( A \) or \( B \) that makes \( x_j \) an exact solution.

\( berr \)

Array of size \( \text{nrhs} \times \text{n_err_bnds} \). For each right-hand side, contains information about various error bounds and condition numbers corresponding to the normwise relative error, which is defined as follows:

Normwise relative error in the \( i \)-th solution vector

\[ \frac{\max_j |X_{true,j} - X_{i,j}|}{\max_j |X_{i,j}|} \]

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned.

\( err\_bnds\_norm \)

- \( err = 1 \)
  - "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for \texttt{chesvxx} and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for \texttt{zhesvxx}.

- \( err = 2 \)
  - "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for \texttt{chesvxx} and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for \texttt{zhesvxx}. This error bound should only be trusted if the previous boolean is true.

- \( err = 3 \)
  - Reciprocal condition number. Estimated normwise reciprocal condition number. Compared with the threshold \( \sqrt{n} \cdot \text{slamch}(\varepsilon) \) for \texttt{chesvxx} and \( \sqrt{n} \cdot \text{dlamch}(\varepsilon) \) for \texttt{zhesvxx} to determine if
the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$

Let $z=s^*a$, where $s$ scales each row by a power of the radix so all absolute row sums of $z$ are approximately 1.

The information for right-hand side $i$, where $1 \leq i \leq \text{nrhs}$, and type of error $\text{err}$ is stored in $\text{err_bnds_norm[ ( (\text{err}-1) \* \text{nrhs} + i - 1 )]}$.

Array of size $\text{nrhs} \* \text{n_err_bnds}$. For each right-hand side, contains information about various error bounds and condition numbers corresponding to the componentwise relative error, which is defined as follows:

Componentwise relative error in the $i$-th solution vector:

$$\max \frac{|X_{true,j_i} - X_{j_i}|}{|X_{j_i}|}$$

The array is indexed by the type of error information as described below. There are currently up to three pieces of information returned for each right-hand side. If componentwise accuracy is not requested ($\text{params[2]} = 0.0$), then $\text{err_bnds_comp}$ is not accessed.

$\text{err}=1$ "Trust/don't trust" boolean. Trust the answer if the reciprocal condition number is less than the threshold $\sqrt{n} \* \text{slamch(\epsilon)}$ for chesvxx and $\sqrt{n} \* \text{dlamch(\epsilon)}$ for zhesvxx.

$\text{err}=2$ "Guaranteed" error bound. The estimated forward error, almost certainly within a factor of 10 of the true error so long as the next entry is greater than the threshold $\sqrt{n} \* \text{slamch(\epsilon)}$ for chesvxx and $\sqrt{n} \* \text{dlamch(\epsilon)}$ for zhesvxx. This error bound should only be trusted if the previous boolean is true.

$\text{err}=3$ Reciprocal condition number. Estimated componentwise reciprocal condition number. Compared with the threshold $\sqrt{n} \* \text{slamch(\epsilon)}$ for chesvxx and $\sqrt{n} \* \text{dlamch(\epsilon)}$ for zhesvxx to determine if the error estimate is "guaranteed". These reciprocal condition numbers for some appropriately scaled matrix $Z$ are:

$$\|Z\|_\infty \cdot \|Z^{-1}\|_\infty$$
Let \( z = s \cdot (a \cdot \text{diag}(x)) \), where \( x \) is the solution for the current right-hand side and \( s \) scales each row of \( a \cdot \text{diag}(x) \) by a power of the radix so all absolute row sums of \( z \) are approximately 1.

The information for right-hand side \( i \), where \( 1 \leq i \leq \text{nrhs} \), and type of error \( \text{err} \) is stored in \( \text{err}_\text{bnds}_\text{comp}[(\text{err}-1)\times\text{nrhs} + i - 1] \).

\( \text{ipiv} \)

If \( \text{fact} = 'N' \), \( \text{ipiv} \) is an output argument and on exit contains details of the interchanges and the block structure \( D \), as determined by \text{ssytrf} for single precision flavors and \text{dsytrf} for double precision flavors.

\( \text{equed} \)

If \( \text{fact} ≠ 'F' \), then \( \text{equed} \) is an output argument. It specifies the form of equilibration that was done (see the description of \( \text{equed} \) in Input Arguments section).

\( \text{params} \)

If an entry is less than 0.0, that entry is filled with the default value used for that parameter, otherwise the entry is not modified.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful. The solution to every right-hand side is guaranteed.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( 0 < \text{info}; \text{U}_\text{info}.\text{info} \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, so the solution and error bounds could not be computed; \( rcond = 0 \) is returned.

If \( \text{info} = n+j \); The solution corresponding to the \( j \)-th right-hand side is not guaranteed. The solutions corresponding to other right-hand sides \( k \) with \( k > j \) may not be guaranteed as well, but only the first such right-hand side is reported. If a small componentwise error is not requested \( \text{params}[2] = 0.0 \), then the \( j \)-th right-hand side is the first with a normwise error bound that is not guaranteed (the smallest \( j \) such that for column major layout \( \text{err}_\text{bnds}_\text{norm}[j - 1] = 0.0 \) or \( \text{err}_\text{bnds}_\text{comp}[j - 1] = 0.0 \); or for row major layout \( \text{err}_\text{bnds}_\text{norm}[(j - 1)\times\text{n_err_bnds}] = 0.0 \) or \( \text{err}_\text{bnds}_\text{comp}[(j - 1)\times\text{n_err_bnds}] = 0.0 \). See the definition of \( \text{err}_\text{bnds}_\text{norm} \) and \( \text{err}_\text{bnds}_\text{comp} \) for \( \text{err} = 1 \). To get information about all of the right-hand sides, check \( \text{err}_\text{bnds}_\text{norm} \) or \( \text{err}_\text{bnds}_\text{comp} \).

**See Also**

Matrix Storage Schemes

\( ? \text{spsv} \)

*Computes the solution to the system of linear equations with a real or complex symmetric coefficient matrix \( A \) stored in packed format, and multiple right-hand sides.*

**Syntax**

```c
lapack_int LAPACKE_sspsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , float * ap , lapack_int * ipiv , float * b , lapack_int ldb );

lapack_int LAPACKE_dspsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , double * ap , lapack_int * ipiv , double * b , lapack_int ldb );

lapack_int LAPACKE_cspsv (int matrix_layout , char uplo , lapack_int n , lapack_int nrhs , lapack_complex_float * ap , lapack_int * ipiv , lapack_complex_float * b , lapack_int ldb );
```
LAPACK Routines

lapack_int LAPACKE_zspsv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * ap, lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);

Include Files
• mkl.h

Description

The routine solves for $X$ the real or complex system of linear equations $A*X = B$, where $A$ is an $n$-by-$n$ symmetric matrix stored in packed format, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

The diagonal pivoting method is used to factor $A$ as $A = U*D*U^T$ or $A = L*D*L^T$, where $U$ (or $L$) is a product of permutation and unit upper (lower) triangular matrices, and $D$ is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of $A$ is then used to solve the system of equations $A*X = B$.

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix_layout</td>
<td>Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).</td>
</tr>
<tr>
<td>uplo</td>
<td>Must be 'U' or 'L'.</td>
</tr>
<tr>
<td></td>
<td>Indicates whether the upper or lower triangular part of $A$ is stored:</td>
</tr>
<tr>
<td></td>
<td>If uplo = 'U', the upper triangle of $A$ is stored.</td>
</tr>
<tr>
<td></td>
<td>If uplo = 'L', the lower triangle of $A$ is stored.</td>
</tr>
<tr>
<td>n</td>
<td>The order of matrix $A$; $n \geq 0$.</td>
</tr>
<tr>
<td>nrhs</td>
<td>The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.</td>
</tr>
<tr>
<td>ap, b</td>
<td>Arrays: $ap$ (size max(1,$n*(n+1)/2$), $b$ of size max(1,$ldb<em>nrhs$) for column major layout and max(1,$ldb</em>n$) for row major layout.</td>
</tr>
<tr>
<td></td>
<td>The array $ap$ contains the factor $U$ or $L$, as specified by $uplo$, in packed storage (see Matrix Storage Schemes).</td>
</tr>
<tr>
<td></td>
<td>The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.</td>
</tr>
<tr>
<td>ldb</td>
<td>The leading dimension of $b$; $ldb \geq \max(1,n)$ for column major layout and $ldb \geq nrhs$ for row major layout.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ap</td>
<td>The block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$) from the factorization of $A$ as computed by $?sptrf$, stored as a packed triangular matrix in the same storage format as $A$.</td>
</tr>
<tr>
<td>b</td>
<td>If $info = 0$, $b$ is overwritten by the solution matrix $X$.</td>
</tr>
<tr>
<td>ipiv</td>
<td>Array, size at least max(1, $n$). Contains details of the interchanges and the block structure of $D$, as determined by $?sptrf$.</td>
</tr>
</tbody>
</table>
If $\text{ipiv}[i-1] = k > 0$, then $d_{ij}$ is a 1-by-1 block, and the $i$-th row and column of $A$ was interchanged with the $k$-th row and column.

If $\text{uplo} = 'U$' and $\text{ipiv}[i]=\text{ipiv}[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $i$-th row and column of $A$ was interchanged with the $m$-th row and column.

If $\text{uplo} = 'L$' and $\text{ipiv}[i-1] = \text{ipiv}[i] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $(i+1)$-th row and column of $A$ was interchanged with the $m$-th row and column.

**Return Values**

This function returns a value $\text{info}$.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, parameter $i$ had an illegal value.

If $\text{info} = i$, $d_{ii}$ is 0. The factorization has been completed, but $D$ is exactly singular, so the solution could not be computed.

**See Also**

Matrix Storage Schemes

?spsvx

*Uses the diagonal pivoting factorization to compute the solution to the system of linear equations with a real or complex symmetric coefficient matrix $A$ stored in packed format, and provides error bounds on the solution.*

**Syntax**

```c
lapack_int LAPACKE_spsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const float* ap, float* afp, lapack_int* ipiv, const float* b,
lapack_int ldb, float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_dpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const double* ap, double* afp, lapack_int* ipiv, const double* b,
lapack_int ldb, double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

lapack_int LAPACKE_cspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int* ipiv,
const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x,
lapack_int ldx, float* rcond, float* ferr, float* berr );

lapack_int LAPACKE_zspsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );
```

**Include Files**

- mkl.h

**Description**
The routine uses the diagonal pivoting factorization to compute the solution to a real or complex system of linear equations $A*X = B$, where $A$ is a $n$-by-$n$ symmetric matrix stored in packed format, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine ?spsvx performs the following steps:

1. If fact = 'N', the diagonal pivoting method is used to factor the matrix $A$. The form of the factorization is $A = U*D*U^T$ or $A = L*D*L^T$, where $U$ (or $L$) is a product of permutation and unit upper (lower) triangular matrices, and $D$ is symmetric and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

2. If some $d_{ii}= 0$, so that $D$ is exactly singular, then the routine returns with info = $i$. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, info = $n+1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.

3. The system of equations is solved for $X$ using the factored form of $A$.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **fact**: Must be 'F' or 'N'.
  
  Specifies whether or not the factored form of the matrix $A$ has been supplied on entry.

  If fact = 'F': on entry, afp and ipiv contain the factored form of $A$. Arrays $ap$, afp, and ipiv are not modified.

  If fact = 'N', the matrix $A$ is copied to afp and factored.

- **uplo**: Must be 'U' or 'L'.
  
  Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:

  If uplo = 'U', the array $ap$ stores the upper triangular part of the symmetric matrix $A$, and $A$ is factored as $U*D*U^T$.

  If uplo = 'L', the array $ap$ stores the lower triangular part of the symmetric matrix $A$; $A$ is factored as $L*D*L^T$.

- **n**: The order of matrix $A$; $n \geq 0$.

- **nrhs**: The number of right-hand sides, the number of columns in $B$; $nrhs \geq 0$.

- **ap, afp, b**: Arrays: $ap$ (size max(1,$n*(n+1)/2$), afp (size max(1,$n*(n+1)/2$), bof size max(1, $ldb*nrhs$) for column major layout and max(1, $ldb*n$) for row major layout.

  The array $ap$ contains the upper or lower triangle of the symmetric matrix $A$ in packed storage (see Matrix Storage Schemes).
The array `afp` is an input argument if `fact = 'F'`. It contains the block diagonal matrix `D` and the multipliers used to obtain the factor `U` or `L` from the factorization `A = U*D*U^T` or `A = L*D*L^T` as computed by `?sptrf`, in the same storage format as `A`.

The array `b` contains the matrix `B` whose columns are the right-hand sides for the systems of equations.

`ldb` The leading dimension of `b`; `ldb ≥ max(1, n)` for column major layout and `ldb≥nrhs` for row major layout.

`ipiv` Array, size at least `max(1, n)`. The array `ipiv` is an input argument if `fact = 'F'`. It contains details of the interchanges and the block structure of `D`, as determined by `?sptrf`.

If `ipiv[i-1] = k > 0`, then `d_{ii}` is a 1-by-1 block, and the `i`-th row and column of `A` was interchanged with the `k`-th row and column.

If `uplo = 'U'` and `ipiv[i]=ipiv[i-1] = -m < 0`, then `D` has a 2-by-2 block in rows/columns `i` and `i+1`, and `i`-th row and column of `A` was interchanged with the `m`-th row and column.

If `uplo = 'L'` and `ipiv[i-1] =ipiv[i] = -m < 0`, then `D` has a 2-by-2 block in rows/columns `i` and `i+1`, and `(i+1)`-th row and column of `A` was interchanged with the `m`-th row and column.

`ldx` The leading dimension of the output array `x`; `ldx≥ max(1, n)` for column major layout and `ldx≥nrhs` for row major layout.

Output Parameters

`x` Array, size `max(1, ldx*nrhs)` for column major layout and `max(1, ldx*n)` for row major layout.

If `info = 0` or `info = n+1`, the array `x` contains the solution matrix `X` to the system of equations.

`afp, ipiv` These arrays are output arguments if `fact = 'N'`. See the description of `afp, ipiv in Input Arguments` section.

`rcond` An estimate of the reciprocal condition number of the matrix `A`. If `rcond` is less than the machine precision (in particular, if `rcond = 0`), the matrix is singular to working precision. This condition is indicated by a return code of `info > 0`.

`ferr, berr` Arrays, size at least `max(1, nrhs)`. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info = -i`, parameter `i` had an illegal value.

If `info = i`, and `i≤n`, then `d_{ii}` is exactly zero. The factorization has been completed, but the block diagonal matrix `D` is exactly singular, so the solution and error bounds could not be computed; `rcond = 0` is returned.
If $info = i$, and $i = n + 1$, then $D$ is nonsingular, but $rcond$ is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of $rcond$ would suggest.

See Also
Matrix Storage Schemes

?hpsv
Computation the solution to the system of linear equations with a Hermitian coefficient matrix $A$ stored in packed format, and multiple right-hand sides.

Syntax

```c
lapack_int LAPACKE_chpsv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * ap, lapack_int * ipiv, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_zhpsv (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * ap, lapack_int * ipiv, lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h

Description

The routine solves for $X$ the system of linear equations $A*X = B$, where $A$ is an $n$-by-$n$ Hermitian matrix stored in packed format, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

The diagonal pivoting method is used to factor $A$ as $A = U*D*U^H$ or $A = L*D*L^H$, where $U$ (or $L$) is a product of permutation and unit upper (lower) triangular matrices, and $D$ is Hermitian and block diagonal with 1-by-1 and 2-by-2 diagonal blocks.

The factored form of $A$ is then used to solve the system of equations $A*X = B$.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo` Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of $A$ is stored:
  - If `uplo` = 'U', the upper triangle of $A$ is stored.
  - If `uplo` = 'L', the lower triangle of $A$ is stored.
- `n` The order of matrix $A$; $n \geq 0$.
- `nrhs` The number of right-hand sides; the number of columns in $B$; $nrhs \geq 0$.
- `ap`, `b` Arrays: `ap` (size max(1,$n*(n+1)/2$), bof size max(1, $ldb*nrhs$)) for column major layout and max(1, $ldb*n$) for row major layout.
The array `ap` contains the factor $U$ or $L$, as specified by `uplo`, in packed storage (see Matrix Storage Schemes).

The array `b` contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

`ldb` The leading dimension of `b`; `ldb` ≥ max(1, `n`) for column major layout and `ldb` ≥ nrhs for row major layout.

### Output Parameters

**ap**

The block-diagonal matrix $D$ and the multipliers used to obtain the factor $U$ (or $L$) from the factorization of $A$ as computed by ?hptrf, stored as a packed triangular matrix in the same storage format as $A$.

**b**

If `info` = 0, `b` is overwritten by the solution matrix $X$.

**ipiv**

Array, size at least max(1, `n`). Contains details of the interchanges and the block structure of $D$, as determined by ?hptrf.

If `ipiv[i-1]` = `k` > 0, then $d_{ii}$ is a 1-by-1 block, and the $i$-th row and column of $A$ was interchanged with the $k$-th row and column.

If `uplo` = 'U' and `ipiv[i]`=`ipiv[i-1]` = `-m` < 0, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $i$-th row and column of $A$ was interchanged with the $m$-th row and column.

If `uplo` = 'L' and `ipiv[i-1]`=`ipiv[i]` = `-m` < 0, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and ($i+1$)-th row and column of $A$ was interchanged with the $m$-th row and column.

### Return Values

This function returns a value `info`.

If `info` = 0, the execution is successful.

If `info` = `-i`, parameter `i` had an illegal value.

If `info` = `i`, $d_{ii}$ is 0. The factorization has been completed, but $D$ is exactly singular, so the solution could not be computed.

### See Also

Matrix Storage Schemes

### ?hpsvx

*Use the diagonal pivoting factorization to compute the solution to the system of linear equations with a Hermitian coefficient matrix $A$ stored in packed format, and provides error bounds on the solution.*

### Syntax

```c
lapack_int LAPACKE_chpsvx( int matrix_layout, char fact, char uplo, lapack_int n, lapack_int nrhs, const lapack_complex_float* ap, lapack_complex_float* afp, lapack_int* ipiv, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* x, lapack_int ldx, float* rcond, float* ferr, float* berr );
```
lapack_int LAPACKE_zhpsvx( int matrix_layout, char fact, char uplo, lapack_int n,
lapack_int nrhs, const lapack_complex_double* ap, lapack_complex_double* afp,
lapack_int* ipiv, const lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* x, lapack_int ldx, double* rcond, double* ferr, double* berr );

**Include Files**
- mkl.h

**Description**
The routine uses the diagonal pivoting factorization to compute the solution to a complex system of linear equations $A \times X = B$, where $A$ is an $n$-by-$n$ Hermitian matrix stored in packed format, the columns of matrix $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.

Error bounds on the solution and a condition estimate are also provided.

The routine `zhpsvx` performs the following steps:

1. If `fact = 'N'`, the diagonal pivoting method is used to factor the matrix $A$. The form of the factorization is $A = U \times D \times U^H$ or $A = L \times D \times L^H$, where $U$ (or $L$) is a product of permutation and unit upper (lower) triangular matrices, and $D$ is a Hermitian and block diagonal with $1$-by-$1$ and $2$-by-$2$ diagonal blocks.

2. If some $d_{ij} = 0$, so that $D$ is exactly singular, then the routine returns with $info = i$. Otherwise, the factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than machine precision, $info = n+1$ is returned as a warning, but the routine still goes on to solve for $X$ and compute error bounds as described below.

3. The system of equations is solved for $X$ using the factored form of $A$.

4. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **fact**
  Must be 'F' or 'N'.
  Specifies whether or not the factored form of the matrix $A$ has been supplied on entry.
  If `fact = 'F'`: on entry, $afp$ and $ipiv$ contain the factored form of $A$. Arrays $ap$, $afp$, and $ipiv$ are not modified.
  If `fact = 'N'`, the matrix $A$ is copied to $afp$ and factored.

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factored:
  - If `uplo = 'U'`, the array $ap$ stores the upper triangular part of the Hermitian matrix $A$, and $A$ is factored as $U \times D \times U^H$.
  - If `uplo = 'L'`, the array $ap$ stores the lower triangular part of the Hermitian matrix $A$, and $A$ is factored as $L \times D \times L^H$.

- **n**
  The order of matrix $A$; $n \geq 0$. 

---

719
nrhs
The number of right-hand sides, the number of columns in B; $nrhs \geq 0$.

$ap, afp, b$
Arrays: $ap$ (size max(1, $n^*$(n+1)/2)), $afp$ (size max(1, $n^*$(n+1)/2)), $b$ of size max(1, $ldb^*nrhs$) for column major layout and max(1, $ldb^*n$) for row major layout.

The array $ap$ contains the upper or lower triangle of the Hermitian matrix $A$ in packed storage (see Matrix Storage Schemes).

The array $afp$ is an input argument if $fact = 'F'$. It contains the block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ or $L$ from the factorization $A = U^D U^H$ or $A = L^D L^H$ as computed by ?hptrf, in the same storage format as $A$.

The array $b$ contains the matrix $B$ whose columns are the right-hand sides for the systems of equations.

ldb
The leading dimension of $b$; $ldb \geq \max(1, n)$ for column major layout and $ldb \geq nrhs$ for row major layout.

ipiv
Array, size at least $\max(1, n)$. The array $ipiv$ is an input argument if $fact = 'F'$. It contains details of the interchanges and the block structure of $D$, as determined by ?hptrf.

If $ipiv[i-1] = k > 0$, then $d_{ii}$ is a 1-by-1 block, and the $i$-th row and column of $A$ was interchanged with the $k$-th row and column.

If $uplo = 'U'$ and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $i$-th row and column of $A$ was interchanged with the $m$-th row and column.

If $uplo = 'L'$ and $ipiv[i] = ipiv[i-1] = -m < 0$, then $D$ has a 2-by-2 block in rows/columns $i$ and $i+1$, and $(i+1)$-th row and column of $A$ was interchanged with the $m$-th row and column.

ldx
The leading dimension of the output array $x$; $ldx \geq \max(1, n)$ for column major layout and $ldx \geq nrhs$ for row major layout.

Output Parameters

$x$
Array, size max(1, $ldx^*nrhs$) for column major layout and max(1, $ldx^*n$) for row major layout.

If $info = 0$ or $info = n+1$, the array $x$ contains the solution matrix $X$ to the system of equations.

$afp, ipiv$
These arrays are output arguments if $fact = 'N'$. See the description of $afp, ipiv$ in Input Arguments section.

rcond
An estimate of the reciprocal condition number of the matrix $A$. If $rcond$ is less than the machine precision (in particular, if $rcond = 0$), the matrix is singular to working precision. This condition is indicated by a return code of $info > 0$.

ferr
Array, size at least max(1, $nrhs$). Contains the estimated forward error bound for each solution vector $x_j$ (the $j$-th column of the solution matrix $X$). If $xtrue$ is the true solution corresponding to $x_j$, $ferr[j-1]$
is an estimated upper bound for the magnitude of the largest element in \((x_j - x_{true})\) divided by the magnitude of the largest element in \(x_j\).
The estimate is as reliable as the estimate for \(rcond\), and is almost always a slight overestimate of the true error.

**Return Values**

This function returns a value \(info\). If \(info = 0\), the execution is successful. If \(info = -i\), parameter \(i\) had an illegal value.

If \(info = i\), and \(i \leq n\), then \(d_{ii}\) is exactly zero. The factorization has been completed, but the block diagonal matrix \(D\) is exactly singular, so the solution and error bounds could not be computed; \(rcond = 0\) is returned.

If \(info = i\), and \(i = n + 1\), then \(D\) is nonsingular, but \(rcond\) is less than machine precision, meaning that the matrix is singular to working precision. Nevertheless, the solution and error bounds are computed because there are a number of situations where the computed solution can be more accurate than the value of \(rcond\) would suggest.

**See Also**

Matrix Storage Schemes

**LAPACK Least Squares and Eigenvalue Problem Routines**

This section includes descriptions of LAPACK computational routines and driver routines for solving linear least squares problems, eigenvalue and singular value problems, and performing a number of related computational tasks. For a full reference on LAPACK routines and related information see [LUG].

**Least Squares Problems.** A typical least squares problem is as follows: given a matrix \(A\) and a vector \(b\), find the vector \(x\) that minimizes the sum of squares \(\sum_i ((Ax)_i - b_i)^2\) or, equivalently, find the vector \(x\) that minimizes the 2-norm \(||Ax - b||_2\).

In the most usual case, \(A\) is an \(m\)-by-\(n\) matrix with \(m \geq n\) and \(\text{rank}(A) = n\). This problem is also referred to as finding the least squares solution to an overdetermined system of linear equations (here we have more equations than unknowns). To solve this problem, you can use the QR factorization of the matrix \(A\) (see QR Factorization).

If \(m < n\) and \(\text{rank}(A) = m\), there exist an infinite number of solutions \(x\) which exactly satisfy \(Ax = b\), and thus minimize the norm \(||Ax - b||_2\). In this case it is often useful to find the unique solution that minimizes \(||x||_2\). This problem is referred to as finding the minimum-norm solution to an underdetermined system of linear equations (here we have more unknowns than equations). To solve this problem, you can use the LQ factorization of the matrix \(A\) (see LQ Factorization).

In the general case you may have a rank-deficient least squares problem, with \(\text{rank}(A) < \min(m, n)\); find the minimum-norm least squares solution that minimizes both \(||x||_2\) and \(||Ax - b||_2\). In this case (or when the rank of \(A\) is in doubt) you can use the QR factorization with pivoting or singular value decomposition (see Singular Value Decomposition).

**Eigenvalue Problems.** The eigenvalue problems (from German "eigen" "own") are stated as follows: given a matrix \(A\), find the eigenvalues \(\lambda\) and the corresponding eigenvectors \(z\) that satisfy the equation \(Az = \lambda z\) (right eigenvectors \(z\)) or the equation...
\( z^H A = \lambda z^H \) (left eigenvectors \( z \)).

If \( A \) is a real symmetric or complex Hermitian matrix, the above two equations are equivalent, and the problem is called a symmetric eigenvalue problem. Routines for solving this type of problems are described in the section Symmetric Eigenvalue Problems.

Routines for solving eigenvalue problems with nonsymmetric or non-Hermitian matrices are described in the section Nonsymmetric Eigenvalue Problems.

The library also includes routines that handle generalized symmetric-definite eigenvalue problems: find the eigenvalues \( \lambda \) and the corresponding eigenvectors \( x \) that satisfy one of the following equations:

\[ Az = \lambda Bz, \quad ABz = \lambda z, \quad \text{or} \quad BAz = \lambda z, \]

where \( A \) is symmetric or Hermitian, and \( B \) is symmetric positive-definite or Hermitian positive-definite. Routines for reducing these problems to standard symmetric eigenvalue problems are described in the section Generalized Symmetric-Definite Eigenvalue Problems.

To solve a particular problem, you usually call several computational routines. Sometimes you need to combine the routines of this chapter with other LAPACK routines described in "LAPACK Routines: Linear Equations" as well as with BLAS routines described in "BLAS and Sparse BLAS Routines".

For example, to solve a set of least squares problems minimizing \( \| Ax - b \|^2 \) for all columns \( b \) of a given matrix \( B \) (where \( A \) and \( B \) are real matrices), you can call \( ?geqrf \) to form the factorization \( A = QR \), then call \( ?ormqr \) to compute \( C = Q^H B \) and finally call the BLAS routine \( ?trsm \) to solve for \( X \) the system of equations \( RX = C \).

Another way is to call an appropriate driver routine that performs several tasks in one call. For example, to solve the least squares problem the driver routine \( ?gels \) can be used.

**LAPACK Least Squares and Eigenvalue Problem Computational Routines**

In the sections that follow, the descriptions of LAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- Orthogonal Factorizations
- Singular Value Decomposition
- Symmetric Eigenvalue Problems
- Generalized Symmetric-Definite Eigenvalue Problems
- Nonsymmetric Eigenvalue Problems
- Generalized Nonsymmetric Eigenvalue Problems
- Generalized Singular Value Decomposition

See also the respective driver routines.

**Orthogonal Factorizations: LAPACK Computational Routines**

This section describes the LAPACK routines for the \( QR (RQ) \) and \( LQ (QL) \) factorization of matrices. Routines for the \( RZ \) factorization as well as for generalized \( QR \) and \( RQ \) factorizations are also included.

**QR Factorization.** Assume that \( A \) is an \( m \)-by-\( n \) matrix to be factored.

If \( m \geq n \), the \( QR \) factorization is given by

\[
A = Q \begin{pmatrix} R \\ 0 \end{pmatrix} = \begin{pmatrix} Q_1, \ Q_2 \end{pmatrix} \begin{pmatrix} R \\ 0 \end{pmatrix}
\]

where \( R \) is an \( n \)-by-\( n \) upper triangular matrix with real diagonal elements, and \( Q \) is an \( m \)-by-\( m \) orthogonal (or unitary) matrix.
You can use the QR factorization for solving the following least squares problem: minimize $||Ax - b||^2$ where $A$ is a full-rank $m$-by-$n$ matrix ($m \geq n$). After factoring the matrix, compute the solution $x$ by solving $Rx = (Q_1)^\top b$.

If $m < n$, the QR factorization is given by

$$A = QR = Q(R_1R_2)$$

where $R$ is trapezoidal, $R_1$ is upper triangular and $R_2$ is rectangular.

$Q$ is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with $Q$ in this representation.

**LQ Factorization** LQ factorization of an $m$-by-$n$ matrix $A$ is as follows. If $m \leq n$,

$$A = \begin{pmatrix} L & 0 \end{pmatrix} Q = \begin{pmatrix} L & 0 \end{pmatrix} \begin{pmatrix} Q_1 \\ Q_2 \end{pmatrix} = \begin{pmatrix} LQ_1 \end{pmatrix}$$

where $L$ is an $m$-by-$m$ lower triangular matrix with real diagonal elements, and $Q$ is an $n$-by-$n$ orthogonal (or unitary) matrix.

If $m > n$, the LQ factorization is

$$A = \begin{pmatrix} L_1 \\ L_2 \end{pmatrix} Q$$

where $L_1$ is an $n$-by-$n$ lower triangular matrix, $L_2$ is rectangular, and $Q$ is an $n$-by-$n$ orthogonal (or unitary) matrix.

You can use the LQ factorization to find the minimum-norm solution of an underdetermined system of linear equations $Ax = b$ where $A$ is an $m$-by-$n$ matrix of rank $m$ ($m < n$). After factoring the matrix, compute the solution vector $x$ as follows: solve $Ly = b$ for $y$, and then compute $x = (Q_1)^\top y$.

Table "Computational Routines for Orthogonal Factorization" lists LAPACK routines that perform orthogonal factorization of matrices.

**Computational Routines for Orthogonal Factorization**

<table>
<thead>
<tr>
<th>Matrix type, factorization</th>
<th>Factorize without pivoting</th>
<th>Factorize with pivoting</th>
<th>Generate matrix Q</th>
<th>Apply matrix Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>general matrices, QR factorization</td>
<td>geqrf</td>
<td>geqpf</td>
<td>orgqr</td>
<td>ormrq</td>
</tr>
<tr>
<td></td>
<td>geqrfp</td>
<td>geqp3</td>
<td>ungqr</td>
<td>unmqr</td>
</tr>
<tr>
<td>general matrices, blocked QR factorization</td>
<td>geqrt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>general matrices, RQ factorization</td>
<td>gerqf</td>
<td></td>
<td>orgrq</td>
<td>ormrq</td>
</tr>
<tr>
<td></td>
<td>ungrq</td>
<td></td>
<td>ormlq</td>
<td></td>
</tr>
<tr>
<td>general matrices, LQ factorization</td>
<td>gelqf</td>
<td></td>
<td>orglq</td>
<td>ormlq</td>
</tr>
<tr>
<td></td>
<td>ungql</td>
<td></td>
<td>unmlq</td>
<td></td>
</tr>
<tr>
<td>general matrices, QL factorization</td>
<td>geqlf</td>
<td></td>
<td>orgql</td>
<td>ormlq</td>
</tr>
<tr>
<td></td>
<td>ungql</td>
<td></td>
<td>unmlq</td>
<td></td>
</tr>
<tr>
<td>trapezoidal matrices, RZ factorization</td>
<td>tizrzf</td>
<td></td>
<td></td>
<td>ormrz</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>unmrz</td>
</tr>
<tr>
<td>pair of matrices, generalized QR factorization</td>
<td>ggqrq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix type, factorization</td>
<td>Factorize without pivoting</td>
<td>Factorize with pivoting</td>
<td>Generate matrix Q</td>
<td>Apply matrix Q</td>
</tr>
<tr>
<td>---------------------------------------------------------------</td>
<td>-----------------------------</td>
<td>-------------------------</td>
<td>-------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>pair of matrices, generalized RQ factorization</td>
<td>ggrqf</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular-pentagonal matrices, blocked QR factorization</td>
<td>tpqrt</td>
<td>tpmqrt</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`?geqrf`

*Computes the QR factorization of a general m-by-n matrix.*

**Syntax**

```c
lapack_int LAPACKE_sgeqrf (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);
lapack_int LAPACKE_dgeqrf (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);
lapack_int LAPACKE_cgeqrf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
lapack_int LAPACKE_zgeqrf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

**Include Files**

- mkl.h

**Description**

The routine forms the QR factorization of a general m-by-n matrix \( A \) (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix \( Q \) explicitly. Instead, \( Q \) is represented as a product of \( \min(m, n) \) elementary reflectors. Routines are provided to work with \( Q \) in this representation.

**NOTE**

This routine supports the Progress Routine feature. See Progress Function section for details.

**Input Parameters**

- `matrix_layout`: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `m`: The number of rows in the matrix \( A \) (\( m \geq 0 \)).
- `n`: The number of columns in \( A \) (\( n \geq 0 \)).
- `a`: Array \( a \) of size \( \max(1, lda*n) \) for column major layout and \( \max(1, lda*m) \) for row major layout contains the matrix \( A \).
- `lda`: The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and at least \( \max(1, n) \) for row major layout.
Output Parameters

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the \( \min(m, n) \) -by-\( n \) upper trapezoidal matrix \( R \) (\( R \) is upper triangular if \( m \geq n \)); the elements below the diagonal, with the array \( \tau \), present the orthogonal matrix \( Q \) as a product of \( \min(m, n) \) elementary reflectors (see Orthogonal Factorizations).

\( \tau \)

Array, size at least \( \text{max}(1, \min(m, n)) \). Contains scalars that define elementary reflectors for the matrix \( Q \) in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix \( A + E \), where

\[ ||E||_2 = O(\varepsilon) ||A||_2. \]

The approximate number of floating-point operations for real flavors is

\[
\begin{align*}
(4/3)n^3 & \quad \text{if } m = n, \\
(2/3)n^2(3m-n) & \quad \text{if } m > n, \\
(2/3)m^2(3n-m) & \quad \text{if } m < n.
\end{align*}
\]

The number of operations for complex flavors is 4 times greater.

To solve a set of least squares problems minimizing \[ ||A\mathbf{x} - \mathbf{b}\|_2 \] for all columns \( \mathbf{b} \) of a given matrix \( \mathbf{B} \), you can call the following:

\( \text{?geqrf (this routine)} \)

to factorize \( A = QR \);

\( \text{ormqr} \)

to compute \( C = Q^T\mathbf{B} \) (for real matrices);

\( \text{unmqr} \)

to compute \( C = Q^H\mathbf{B} \) (for complex matrices);

\( \text{trsm} \) (a BLAS routine)

to solve \( R\mathbf{X} = \mathbf{C} \).

(The columns of the computed \( \mathbf{X} \) are the least squares solution vectors \( \mathbf{x} \).)

To compute the elements of \( Q \) explicitly, call

\( \text{orgqr} \) \quad \text{(for real matrices)}

\( \text{ungqr} \) \quad \text{(for complex matrices)}.

See Also

\( \text{mkl_progress} \)

Matrix Storage Schemes
?geqrfp
Computes the QR factorization of a general m-by-n matrix with non-negative diagonal elements.

Syntax

```c
lapack_int LAPACKE_sgeqrfp (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);

lapack_int LAPACKE_dgeqrfp (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);

lapack_int LAPACKE_cgeqrfp (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqrfp (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine forms the QR factorization of a general m-by-n matrix A (see Orthogonal Factorizations). No pivoting is performed. The diagonal entries of R are real and nonnegative.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) elementary reflectors. Routines are provided to work with Q in this representation.

**NOTE**

This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `m` The number of rows in the matrix A \((m \geq 0)\).
- `n` The number of columns in A \((n \geq 0)\).
- `a` Array, size max(1,lda*n) for column major layout and max(1,lda*m) for row major layout, containing the matrix A.
- `lda` The leading dimension of a; at least max(1, m) for column major layout and at least max(1, n) for row major layout.

Output Parameters

- `a` Overwritten by the factorization data as follows:
  - The elements on and above the diagonal of the array contain the min(m,n)-by-n upper trapezoidal matrix R (R is upper triangular if m \(\geq n\)); the elements below the diagonal, with the array `tau`, present the orthogonal matrix Q as a product of min(m,n) elementary reflectors (see Orthogonal Factorizations).
  - The diagonal elements of the matrix R are real and non-negative.
tau

Array, size at least max (1, min(m, n)). Contains scalars that define elementary reflectors for the matrix Q in its decomposition in a product of elementary reflectors (see Orthogonal Factorizations).

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes
The computed factorization is the exact factorization of a matrix A + E, where

\[ ||E||_2 = O(\varepsilon) ||A||_2. \]

The approximate number of floating-point operations for real flavors is

\[
\begin{align*}
(4/3)n^3 & \quad \text{if } m = n, \\
(2/3)n^2(3m-n) & \quad \text{if } m > n, \\
(2/3)m^2(3n-m) & \quad \text{if } m < n.
\end{align*}
\]

The number of operations for complex flavors is 4 times greater.
To solve a set of least squares problems minimizing \[ ||A*x - b||_2 \] for all columns b of a given matrix B, you can call the following:

?geqrf (this routine)  
   to factorize \( A = QR; \)
ormqr  
   to compute \( C = Q^T*B \) (for real matrices);
unmqr  
   to compute \( C = QH*B \) (for complex matrices);
trsm (a BLAS routine)  
   to solve \( R*X = C. \)

(The columns of the computed \( X \) are the least squares solution vectors \( x \).)
To compute the elements of \( Q \) explicitly, call

orgqr  
   (for real matrices)
unngqr  
   (for complex matrices).

See Also
mkl_progress

Matrix Storage Schemes

?geqrt  
Computes a blocked QR factorization of a general real or complex matrix using the compact WY representation of Q.

Syntax

```
lapack_int LAPACKE_sgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, float* a, lapack_int lda, float* t, lapack_int ldt);
```
lapack_int LAPACKE_dgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, double* a, lapack_int lda, double* t, lapack_int ldt);
lapack_int LAPACKE_cgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, lapack_complex_float* a, lapack_int lda, lapack_complex_float* t, lapack_int ldt);
lapack_int LAPACKE_zgeqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int nb, lapack_complex_double* a, lapack_int lda, lapack_complex_double* t, lapack_int ldt);

Include Files
- mkl.h

Description
The strictly lower triangular matrix $V$ contains the elementary reflectors $H(i)$ in the $i$th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix $V$ is

$$
V = \begin{bmatrix}
1 \\
v_1 & 1 \\
v_1 & v_2 & 1 \\
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3 \\
\end{bmatrix}
$$

where $v_i$ represents one of the vectors that define $H(i)$. The vectors are returned in the lower triangular part of array $a$.

NOTE
The 1s along the diagonal of $V$ are not stored in $a$.

Let $k = \min(m, n)$. The number of blocks is $b = \text{ceiling}(k/nb)$, where each block is of order $nb$ except for the last block, which is of order $ib = k - (b-1)*nb$. For each of the $b$ blocks, a upper triangular block reflector factor is computed: $t_1, t_2, \ldots, t_b$. The $nb$-by-$nb$ (and $ib$-by-$ib$ for the last block) $t$s are stored in the $nb$-by-$n$ array $t$ as $t = (t_1t_2 \ldots t_b)$.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **m**: The number of rows in the matrix $A$ ($m \geq 0$).
- **n**: The number of columns in $A$ ($n \geq 0$).
- **nb**: The block size to be used in the blocked QR ($\min(m, n) \geq nb \geq 1$).
- **a**: Array $a$ of size max(1, $lda*n$) for column major layout and max(1, $lda*m$) for row major layout contains the $m$-by-$n$ matrix $A$. 


**Output Parameters**

*a*

Overwritten by the factorization data as follows:
The elements on and above the diagonal of the array contain the \( \min(m, n) \)-by-\( n \) upper trapezoidal matrix \( R \) (\( R \) is upper triangular if \( m \geq n \)); the elements below the diagonal, with the array \( t \), present the orthogonal matrix \( Q \) as a product of \( \min(m, n) \) elementary reflectors (see Orthogonal Factorizations).

\( t \)

Array, size \( \max(1, ldt*\min(m, n)) \) for column major layout and \( \max(1, ldt*nb) \) for row major layout.

The upper triangular block reflector's factors stored as a sequence of upper triangular blocks.

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info < 0 \) and \( info = -i \), the \( i \)-th parameter had an illegal value.

**?gemqrt**

Multiplies a general matrix by the orthogonal/unitary matrix \( Q \) of the QR factorization formed by \( ?geqrt \).

**Syntax**

```c
lapack_int LAPACKE_sgemqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int nb, const float* v, lapack_int ldv, const float* t, lapack_int ldt, float* c, lapack_int ldc);

lapack_int LAPACKE_dgemqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int nb, const double* v, lapack_int ldv, const double* t, lapack_int ldt, double* c, lapack_int ldc);

lapack_int LAPACKE_cgemqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int nb, const lapack_complex_float* v, lapack_int ldv, const lapack_complex_float* t, lapack_int ldt, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zgemqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int nb, const lapack_complex_double* v, lapack_int ldv, const lapack_complex_double* t, lapack_int ldt, lapack_complex_double* c, lapack_int ldc);
```

**Include Files**

- mkl.h

**Description**

The \( ?gemqrt \) routine overwrites the general real or complex \( m \)-by-\( n \) matrix \( C \) with
where $Q$ is a real orthogonal (complex unitary) matrix defined as the product of $k$ elementary reflectors

$$Q = H(1) H(2) \ldots H(k) = I - V^T V$$

for real flavors, and

$$Q = H(1) H(2) \ldots H(k) = I - V^H V$$

for complex flavors,

generated using the compact WY representation as returned by geqrt. $Q$ is of order $m$ if $side = 'L'$ and of order $n$ if $side = 'R'$.

### Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **side**:
  - 'L': apply $Q$, $Q^T$, or $Q^H$ from the left.
  - 'R': apply $Q$, $Q^T$, or $Q^H$ from the right.
- **trans**:  
  - 'N', no transpose, apply $Q$.
  - 'T', transpose, apply $Q^T$.
  - 'C', transpose, apply $Q^H$.
- **m**: The number of rows in the matrix $C$, ($m \geq 0$).
- **n**: The number of columns in the matrix $C$, ($n \geq 0$).
- **k**: The number of elementary reflectors whose product defines the matrix $Q$.
- **Constraints:**
  - If $side = 'L'$, $m \geq k \geq 0$
  - If $side = 'R'$, $n \geq k \geq 0$.
- **nb**: The block size used for the storage of $t$, $k \geq nb \geq 1$. This must be the same value of $nb$ used to generate $t$ in geqrt.
- **v**: Array of size max($1, ldv * k$) for column major layout, max($1, ldv * m$) for row major layout and $side = 'L'$, and max($1, ldv * n$) for row major layout and $side = 'R'$.
  The $i$th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \ldots, k$, as returned by geqrt in the first $k$ columns of its array argument $a$.
- **ldv**: The leading dimension of the array $v$.
  - if $side = 'L'$, $ldv$ must be at least max($1, m$) for column major layout and max($1, k$) for row major layout;
  - if $side = 'R'$, $ldv$ must be at least max($1, n$) for column major layout and max($1, k$) for row major layout.
- **t**: Array, size max($1, ldt * \min(m, n)$) for column major layout and max($1, ldt * nb$) for row major layout.
The upper triangular factors of the block reflectors as returned by `geqrt`.

`ldt`

The leading dimension of the array `c`. `ldt` must be at least `nb` for column major layout and `max(1, k)` for row major layout.

`c`

The `m`-by-`n` matrix `C`.

`ldc`

The leading dimension of the array `c`. `ldc` must be at least `max(1, m)` for column major layout and `max(1, n)` for row major layout.

**Output Parameters**

`c`  
Overwritten by the product `Q^*C, C^*Q, Q^T*C, C^*Q^T, Q^H*C, or C^*Q^H` as specified by `side` and `trans`.

**Return Values**

This function returns a value `info`.

If `info=0`, the execution is successful.

If `info = -i`, the `i`-th parameter had an illegal value.

`?geqpf`

*Computes the QR factorization of a general `m`-by-`n` matrix with pivoting.*

**Syntax**

```c
lapack_int LAPACKE_sgeqpf (int matrix_layout, lapack_int m, lapack_int n, float* a,
lapack_int lda, lapack_int* jpvt, float* tau);

lapack_int LAPACKE_dgeqpf (int matrix_layout, lapack_int m, lapack_int n, double* a,
lapack_int lda, lapack_int* jpvt, double* tau);

lapack_int LAPACKE_cgeqpf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, lapack_int* jpvt, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqpf (int matrix_layout, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, lapack_int* jpvt, lapack_complex_double* tau);
```

**Include Files**

- `mkl.h`

**Description**

The routine is deprecated and has been replaced by routine `geqp3`.

The routine `?geqpf` forms the QR factorization of a general `m`-by-`n` matrix `A` with column pivoting: `A*P = Q*R` (see Orthogonal Factorizations). Here `P` denotes an `n`-by-`n` permutation matrix.

The routine does not form the matrix `Q` explicitly. Instead, `Q` is represented as a product of `min(m, n)` elementary reflectors. Routines are provided to work with `Q` in this representation.

**Input Parameters**

`matrix_layout`  
Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).
The number of rows in the matrix $A$ ($n \geq 0$).

The number of columns in $A$ ($n \geq 0$).

Array $a$ of size max(1, $lda \times n$) for column major layout and max(1, $lda \times m$) for row major layout contains the matrix $A$.

The leading dimension of $a$; at least max(1, $m$) for column major layout and max(1, $n$) for row major layout.

Array, size at least max(1, $n$).

On entry, if $jpvt[i - 1] > 0$, the $i$-th column of $A$ is moved to the beginning of $A*P$ before the computation, and fixed in place during the computation.

If $jpvt[i - 1] = 0$, the $i$-th column of $A$ is a free column (that is, it may be interchanged during the computation with any other free column).

Output Parameters

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the min($m,n$)-by-$n$ upper trapezoidal matrix $R$ ($R$ is upper triangular if $m \geq n$); the elements below the diagonal, with the array $tau$, present the orthogonal matrix $Q$ as a product of min($m,n$) elementary reflectors (see Orthogonal Factorizations).

Array, size at least max $(1, \min(m, n))$. Contains additional information on the matrix $Q$.

Overwritten by details of the permutation matrix $P$ in the factorization $A*P = Q*R$. More precisely, the columns of $A*P$ are the columns of $A$ in the following order:

\[ jpvt[0], \, jpvt[1], \, ..., \, jpvt[n - 1]. \]

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

The computed factorization is the exact factorization of a matrix $A + E$, where

\[ \|E\|_2 = O(\varepsilon) \|A\|_2. \]

The approximate number of floating-point operations for real flavors is

\[ (4/3)n^3 \quad \text{if } m = n, \]
\[ (2/3)n^2(3m-n) \quad \text{if } m > n, \]
\[ (2/3)m^2(3n-m) \quad \text{if } m < n. \]

The number of operations for complex flavors is 4 times greater.
To solve a set of least squares problems minimizing \( \| A^*x - b \|_2 \) for all columns \( b \) of a given matrix \( B \), you can call the following:

- \(?\text{geqp}f\) (this routine) to factorize \( A^*P = Q^*R \);
- \(\text{ormqr}\) to compute \( C = Q^T*B \) (for real matrices);
- \(\text{unmqr}\) to compute \( C = Q^H*B \) (for complex matrices);
- \(\text{trsm}\) (a BLAS routine) to solve \( R*X = C \).

(The columns of the computed \( X \) are the permuted least squares solution vectors \( x \); the output array \( \text{jpvt} \) specifies the permutation order.)

To compute the elements of \( Q \) explicitly, call

- \(\text{orgqr}\) (for real matrices)
- \(\text{ungqr}\) (for complex matrices).

\(\text{?\text{geqp}3}\)

\textit{Computes the QR factorization of a general } m \text{-by-} n \text{ matrix with column pivoting using level 3 BLAS.}

**Syntax**

\begin{verbatim}
lapack_int LAPACKE_sgeqp3 (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, lapack_int* jpvt, float* tau);

lapack_int LAPACKE_dgeqp3 (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, lapack_int* jpvt, double* tau);

lapack_int LAPACKE_cgeqp3 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* jpvt, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqp3 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* jpvt, lapack_complex_double* tau);
\end{verbatim}

**Include Files**

- \textit{mkl.h}

**Description**

The routine forms the QR factorization of a general \( m \)-by-\( n \) matrix \( A \) with column pivoting: \( A^*P = Q^*R \) (see Orthogonal Factorizations) using Level 3 BLAS. Here \( P \) denotes an \( n \)-by-\( n \) permutation matrix. Use this routine instead of \text{geqp}f for better performance.

The routine does not form the matrix \( Q \) explicitly. Instead, \( Q \) is represented as a product of \( \min(m, n) \) elementary reflectors. Routines are provided to work with \( Q \) in this representation.

**Input Parameters**

- \textit{matrix_layout}
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \textit{m}
  - The number of rows in the matrix \( A \) \((m \geq 0)\).
- \textit{n}
  - The number of columns in \( A \) \((n \geq 0)\).
Array $a$ of size $\max(1, lda \times n)$ for column major layout and $\max(1, lda \times m)$ for row major layout contains the matrix $A$.

The leading dimension of $a$; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Array, size at least $\max(1, n)$.

On entry, if $jpvt[i - 1] \neq 0$, the $i$-th column of $A$ is moved to the beginning of $AP$ before the computation, and fixed in place during the computation.

If $jpvt[i - 1] = 0$, the $i$-th column of $A$ is a free column (that is, it may be interchanged during the computation with any other free column).

Output Parameters

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the $\min(m,n)$-by-$n$ upper trapezoidal matrix $R$ ($R$ is upper triangular if $m \geq n$); the elements below the diagonal, with the array $tau$, present the orthogonal matrix $Q$ as a product of $\min(m,n)$ elementary reflectors (see Orthogonal Factorizations).

Array, size at least $\max(1, \min(m, n))$. Contains scalar factors of the elementary reflectors for the matrix $Q$.

Overwritten by details of the permutation matrix $P$ in the factorization $A*P = Q*R$. More precisely, the columns of $AP$ are the columns of $A$ in the following order:

$jpvt[0], jpvt[1], ..., jpvt[n - 1]$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

To solve a set of least squares problems minimizing $||A*x - b||_2$ for all columns $b$ of a given matrix $B$, you can call the following:

- `?geqp3` (this routine) to factorize $A*P = Q*R$;
- `ormqr` to compute $C = Q^T*B$ (for real matrices);
- `unmqr` to compute $C = Q^H*B$ (for complex matrices);
- `trsm` (a BLAS routine) to solve $R*X = C$.

(The columns of the computed $X$ are the permuted least squares solution vectors $x$; the output array $jpvt$ specifies the permutation order.)

To compute the elements of $Q$ explicitly, call

- `orgqr` (for real matrices)
ungqr (for complex matrices).

?orgqr
Generates the real orthogonal matrix Q of the QR
factorization formed by ?geqrf.

Syntax
lapack_int LAPACKE_sorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);
lapack_int LAPACKE_dorgqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);

Include Files
• mkl.h

Description
The routine generates the whole or part of m-by-m orthogonal matrix Q of the QR
factorization formed by the routine ?geqrf or geqpf. Use this routine after a call to sgeqrf/dgeqrf or sgeqpf/dgeqpf.

Usually Q is determined from the QR factorization of an m by p matrix A with m≥p. To compute the whole matrix Q, use:
LAPACK_?orgqr(matrix_layout, m, m, p, a, lda, tau)

To compute the leading p columns of Q (which form an orthonormal basis in the space spanned by the
columns of A):
LAPACK_?orgqr(matrix_layout, m, p, p, a, lda)

To compute the matrix Q^k of the QR factorization of leading k columns of the matrix A:
LAPACK_?orgqr(matrix_layout, m, m, k, a, lda, tau)

To compute the leading k columns of Q^k (which form an orthonormal basis in the space spanned by leading k
columns of the matrix A):
LAPACK_?orgqr(matrix_layout, m, k, k, a, lda, tau)

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
or column major (LAPACK_COL_MAJOR).

m
The order of the orthogonal matrix Q (m≥ 0).

n
The number of columns of Q to be computed
(0 ≤ n ≤ m).

k
The number of elementary reflectors whose product defines the matrix Q (0
≤ k ≤ n).

a, tau
Arrays:
a and tau are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf.
The size of $a$ is max(1, $lda*n$) for column major layout and max(1, $lda*m$) for row major layout.

The size of $tau$ must be at least max(1, $k$).

lda

The leading dimension of $a$; at least max(1, $m$) for column major layout and max(1, $n$) for row major layout.

Output Parameters

$a$

Overwritten by $n$ leading columns of the $m$-by-$m$ orthogonal matrix $Q$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

The computed $Q$ differs from an exactly orthogonal matrix by a matrix $E$ such that

$$||E||_2 = O(\varepsilon)||A||_2$$

where $\varepsilon$ is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $n = k$, the number is approximately $(2/3)*n^2*(3m - n)$.

The complex counterpart of this routine is ungqr.

?ormqr

Multiples a real matrix by the orthogonal matrix $Q$ of the QR factorization formed by ?geqrf or ?geqpf.

Syntax

```c
lapack_int LAPACKE_sormqr (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c,
apack_int ldc);

lapack_int LAPACKE_dormqr (int matrix_layout, char side, char trans, lapack_int m,
apack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double* c,
apack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real matrix $C$ by $Q$ or $Q^T$, where $Q$ is the orthogonal matrix $Q$ of the QR factorization formed by the routine ?geqrf or ?geqpf.

Depending on the parameters side left_right and trans, the routine can form one of the matrix products $Q* C$, $Q^T * C$, $C* Q$, or $C^* Q^T$ (overwriting the result on $C$).

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
**side**

Must be either 'L' or 'R'.

If side='L', Q or QT is applied to C from the left.
If side='R', Q or QT is applied to C from the right.

**trans**

Must be either 'N' or 'T'.

If trans='N', the routine multiplies C by Q.
If trans='T', the routine multiplies C by QT.

**m**

The number of rows in the matrix C (m ≥ 0).

**n**

The number of columns in C (n ≥ 0).

**k**

The number of elementary reflectors whose product defines the matrix Q.

Constraints:
0 ≤ k ≤ m if side='L';
0 ≤ k ≤ n if side='R'.

**a, tau, c**

Arrays:
a and tau are the arrays returned by sgeqrf / dgeqrf or sgeqpf / dgeqpf.
The size of a is max(1, lda*k) for column major layout, max(1, lda*m) for row major layout and side = 'L', and max(1, lda*n) for row major layout and side = 'R'.
The size of tau must be at least max(1, k).
Array c of size max(1, ldc*n) for column major layout and max(1, ldc*m) for row major layout contains the m-by-n matrix C.

**lda**

The leading dimension of a. Constraints:
if side = 'L', lda ≥ max(1, m) for column major layout and max(1, k) for row major layout;
if side = 'R', lda ≥ max(1, n) for column major layout and max(1, k) for row major layout.

**ldc**

The leading dimension of c. Constraint:
ldc ≥ max(1, m) for column major layout and max(1, n) for row major layout.

**Output Parameters**

**c**

Overwritten by the product Q*C, Q*T*C, C*Q, or C*Q*(Q as specified by side and trans).

**Return Values**

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

**Application Notes**

The complex counterpart of this routine is unmqr.
Generates the complex unitary matrix $Q$ of the $QR$ factorization formed by \texttt{?geqrf}.

**Syntax**

```c
#include <mkl.h>

lapack_int LAPACKE_cungqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zungqr (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

**Include Files**

- mkl.h

**Description**

The routine generates the whole or part of $m$-by-$m$ unitary matrix $Q$ of the $QR$ factorization formed by the routines \texttt{?geqrf} or \texttt{geqpf}. Use this routine after a call to \texttt{cgeqrf/zgeqrf} or \texttt{cgeqpf/zgeqpf}.

Usually $Q$ is determined from the $QR$ factorization of an $m$ by $p$ matrix $A$ with $m \geq p$. To compute the whole matrix $Q$, use:

```c
LAPACKE_?ungqr(matrix_layout, m, m, p, a, lda, tau)
```

To compute the leading $p$ columns of $Q$ (which form an orthonormal basis in the space spanned by the columns of $A$):

```c
LAPACKE_?ungqr(matrix_layout, m, p, p, a, lda, tau)
```

To compute the matrix $Q^k$ of the $QR$ factorization of the leading $k$ columns of the matrix $A$:

```c
LAPACKE_?ungqr(matrix_layout, m, m, k, a, lda, tau)
```

To compute the leading $k$ columns of $Q^k$ (which form an orthonormal basis in the space spanned by the leading $k$ columns of the matrix $A$):

```c
LAPACKE_?ungqr(matrix_layout, m, k, k, a, lda, tau)
```

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **m**
  The order of the unitary matrix $Q$ ($m \geq 0$).

- **n**
  The number of columns of $Q$ to be computed ($0 \leq n \leq m$).

- **k**
  The number of elementary reflectors whose product defines the matrix $Q$ ($0 \leq k \leq n$).

- **a, tau**
  Arrays: $a$ and $tau$ are the arrays returned by \texttt{cgeqrf/zgeqrf} or \texttt{cgeqpf/zgeqpf}.

  The size of $a$ is max($1, lda*n$) for column major layout and max($1, lda*m$) for row major layout.

  The size of $tau$ must be at least max($1, k$).
The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

**Output Parameters**

\( a \)

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The computed \( Q \) differs from an exactly unitary matrix by a matrix \( E \) such that \( \| E \|_2 = O(\varepsilon) \| A \|_2 \), where \( \varepsilon \) is the machine precision.

The total number of floating-point operations is approximately \( 16*m*n*k - 8*(m + n)*k^2 + (16/3)*k^3 \).

If \( n = k \), the number is approximately \( (8/3)*n^2*(3m - n) \).

The real counterpart of this routine is \text{orgqr}.

\text{?unmqr}

*Multiplies a complex matrix by the unitary matrix \( Q \) of the QR factorization formed by ?geqrf.*

**Syntax**

```c
lapack_int LAPACKE_cunmqr (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmqr (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

**Include Files**

- mkl.h

**Description**

The routine multiplies a rectangular complex matrix \( C \) by \( Q \) or \( Q^H \), where \( Q \) is the unitary matrix \( Q \) of the QR factorization formed by the routines ?geqrf or geqpf.

Depending on the parameters \( \text{side} \) and \( \text{trans} \), the routine can form one of the matrix products \( Q^*C \), \( Q^H*C \), \( C^*Q \), or \( C^*Q^H \) (overwriting the result on \( C \)).

**Input Parameters**

\text{matrix_layout}

Specifies whether matrix storage layout is row major (\text{LAPACK\_ROW\_MAJOR}) or column major (\text{LAPACK\_COL\_MAJOR}).

\text{side}

Must be either \'L\' or \'R\'.

If \( \text{side} = \'L\' \), \( Q \) or \( Q^H \) is applied to \( C \) from the left.

If \( \text{side} = \'R\' \), \( Q \) or \( Q^H \) is applied to \( C \) from the right.
**trans**

Must be either 'N' or 'C'.

If $\text{trans} = 'N'$, the routine multiplies $C$ by $Q$.

If $\text{trans} = 'C'$, the routine multiplies $C$ by $Q^H$.

**m**

The number of rows in the matrix $C$ ($m \geq 0$).

**n**

The number of columns in $C$ ($n \geq 0$).

**k**

The number of elementary reflectors whose product defines the matrix $Q$.

Constraints:

$0 \leq k \leq m$ if $\text{side} = 'L'$;

$0 \leq k \leq n$ if $\text{side} = 'R'$.

**a, c, tau**

Arrays:

$a$ size $\max(1, lda \times k)$ for column major layout, $\max(1, lda \times m)$ for row major layout when $\text{side} = 'L'$, and $\max(1, lda \times n)$ for row major layout when $\text{side} = 'R'$ and $\tau$ are the arrays returned by $\text{cgeqrf} / \text{zgeqrf}$ or $\text{cgeqpf} / \text{zgeqpf}$.

The size of $\tau$ must be at least $\max(1, k)$.

$c$ size $\max(1, ldc \times n)$ for column major layout and $\max(1, ldc \times m)$ for row major layout) contains the $m$-by-$n$ matrix $C$.

**lda**

The leading dimension of $a$. Constraints:

$lda \geq \max(1, m)$ for column major layout and $lda \geq \max(1, k)$ for row major layout if $\text{side} = 'L'$;

$lda \geq \max(1, n)$ for column major layout and $lda \geq \max(1, k)$ for row major layout if $\text{side} = 'R'$.

**ldc**

The leading dimension of $c$. Constraint:

$ldc \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

**Output Parameters**

**c**

Overwritten by the product $Q\times C$, $Q^H\times C$, $C\times Q$, or $C\times Q^H$ (as specified by $\text{side}$ and $\text{trans}$).

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

The real counterpart of this routine is $\text{ormqr}$.

$\text{?gelqf}$

*Computes the LQ factorization of a general $m$-by-$n$ matrix.*
Syntax

```c
lapack_int LAPACKE_sgelqf (int matrix_layout, lapack_int m, lapack_int n, float* a,
                           lapack_int lda, float* tau);

lapack_int LAPACKE_dgelqf (int matrix_layout, lapack_int m, lapack_int n, double* a,
                           lapack_int lda, double* tau);

lapack_int LAPACKE_cgelqf (int matrix_layout, lapack_int m, lapack_int n,
                           lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgelqf (int matrix_layout, lapack_int m, lapack_int n,
                           lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine forms the $LQ$ factorization of a general $m$-by-$n$ matrix $A$ (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix $Q$ explicitly. Instead, $Q$ is represented as a product of $\min(m, n)$ elementary reflectors. Routines are provided to work with $Q$ in this representation.

NOTE
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `m` The number of rows in the matrix $A$ ($m \geq 0$).

- `n` The number of columns in $A$ ($n \geq 0$).

- `a` Array $a$ of size $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the matrix $A$.

- `lda` The leading dimension of $a$; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

- `a` Overwritten by the factorization data as follows:
  The elements on and below the diagonal of the array contain the $m$-by-$\min(m,n)$ lower trapezoidal matrix $L$ ($L$ is lower triangular if $m \leq n$); the elements above the diagonal, with the array `tau`, represent the orthogonal matrix $Q$ as a product of elementary reflectors.

- `tau` Array, size at least $\max(1, \min(m, n))$.
  Contains scalars that define elementary reflectors for the matrix $Q$ (see Orthogonal Factorizations).
Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes
The computed factorization is the exact factorization of a matrix \( A + E \), where
\[
||E||_2 = O(\varepsilon) \quad ||A||_2.
\]
The approximate number of floating-point operations for real flavors is
\[
\begin{align*}
(4/3)n^3 & \quad \text{if } m = n, \\
(2/3)n^2(3m-n) & \quad \text{if } m > n, \\
(2/3)m^2(3n-m) & \quad \text{if } m < n.
\end{align*}
\]
The number of operations for complex flavors is 4 times greater.
To find the minimum-norm solution of an underdetermined least squares problem minimizing \( ||Ax - b||_2 \) for all columns \( b \) of a given matrix \( B \), you can call the following:
- ?gelqf (this routine) to factorize \( A = LQ \);
- trsm (a BLAS routine) to solve \( Ly = B \) for \( Y \);
- ormlq to compute \( X = (Q_1)^T Y \) (for real matrices);
- unmlq to compute \( X = (Q_1)^H Y \) (for complex matrices).
(The columns of the computed \( X \) are the minimum-norm solution vectors \( x \). Here \( A \) is an \( m \)-by-\( n \) matrix with \( m < n \); \( Q_1 \) denotes the first \( m \) columns of \( Q \).
To compute the elements of \( Q \) explicitly, call
- orglq (for real matrices)
- ungq (for complex matrices).

See Also
- mkl_progress

Matrix Storage Schemes

?orglq
Generates the real orthogonal matrix \( Q \) of the \( LQ \) factorization formed by ?gelqf.

Syntax
- \[
\text{lapack_int LAPACKE_sorglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, float* a, lapack_int lda, const float* tau);}
\]
- \[
\text{lapack_int LAPACKE_dorglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, double* a, lapack_int lda, const double* tau);}
\]
Include Files

- mkl.h

Description

The routine generates the whole or part of \( n \)-by-\( n \) orthogonal matrix \( Q \) of the \( LQ \) factorization formed by the routines \texttt{gelqf}. Use this routine after a call to \texttt{sgelqf/dgelqf}.

Usually \( Q \) is determined from the \( LQ \) factorization of an \( p \)-by-\( n \) matrix \( A \) with \( n \geq p \). To compute the whole matrix \( Q \), use:

\[
\text{info} = \text{LAPACKE}_?orglq(matrix\_layout, n, n, p, a, lda, tau)
\]

To compute the leading \( p \) rows of \( Q \), which form an orthonormal basis in the space spanned by the rows of \( A \), use:

\[
\text{info} = \text{LAPACKE}_?orglq(matrix\_layout, p, n, p, a, lda, tau)
\]

To compute the matrix \( Q^k \) of the \( LQ \) factorization of the leading \( k \) rows of \( A \), use:

\[
\text{info} = \text{LAPACKE}_?orglq(matrix\_layout, n, n, k, a, lda, tau)
\]

To compute the leading \( k \) rows of \( Q^k \), which form an orthonormal basis in the space spanned by the leading \( k \) rows of \( A \), use:

\[
\text{info} = \text{LAPACKE}_?orgqr(matrix\_layout, k, n, k, a, lda, tau)
\]

Input Parameters

- \texttt{matrix\_layout}:
  Specifies whether matrix storage layout is row major (\texttt{LAPACK\_ROW\_MAJOR}) or column major (\texttt{LAPACK\_COL\_MAJOR}).

- \texttt{m}:
  The number of rows of \( Q \) to be computed \((0 \leq m \leq n)\).

- \texttt{n}:
  The order of the orthogonal matrix \( Q \) \((n \geq m)\).

- \texttt{k}:
  The number of elementary reflectors whose product defines the matrix \( Q \) \((0 \leq k \leq m)\).

- \texttt{a}, \texttt{tau}:
  Arrays: \texttt{a} \((\text{size max}(1, lda*n) \text{ for column major layout and max}(1, lda*m) \text{ for row major layout})\) and \texttt{tau} are the arrays returned by \texttt{sgelqf/dgelqf}.
  The size of \texttt{tau} must be at least \text{max}(1, k).

- \texttt{lda}:
  The leading dimension of \texttt{a}; at least \text{max}(1, k) for column major layout and \text{max}(1, n) for row major layout.

Output Parameters

- \texttt{a}:
  Overwritten by \( m \) leading rows of the \( n \)-by-\( n \) orthogonal matrix \( Q \).

Return Values

This function returns a value \texttt{info}.

If \texttt{info}=0, the execution is successful.

If \texttt{info} = \(-i\), the \( i\)-th parameter had an illegal value.
Application Notes

The computed $Q$ differs from an exactly orthogonal matrix by a matrix $E$ such that $||E||_2 = O(\varepsilon) * ||A||_2$, where $\varepsilon$ is the machine precision.

The total number of floating-point operations is approximately $4*m*n*k - 2*(m + n)*k^2 + (4/3)*k^3$.

If $m = k$, the number is approximately $(2/3)*m^2*(3n - m)$.

The complex counterpart of this routine is ?unglq.

?ormlq
Multiplies a real matrix by the orthogonal matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```c
lapack_int LAPACKE_sormlq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c, lapack_int ldc);

lapack_int LAPACKE_dormlq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real $m$-by-$n$ matrix $C$ by $Q$ or $Q^T$, where $Q$ is the orthogonal matrix $Q$ of the $LQ$ factorization formed by the routine ?gelqf.

Depending on the parameters side and trans, the routine can form one of the matrix products $Q*C$, $Q^T*C$, $C*Q$, or $C^*Q^T$ (overwriting the result on $C$).

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- side Must be either 'L' or 'R'.
  - If `side` = 'L', $Q$ or $Q^T$ is applied to $C$ from the left.
  - If `side` = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.

- trans Must be either 'N' or 'T'.
  - If `trans` = 'N', the routine multiplies $C$ by $Q$.
  - If `trans` = 'T', the routine multiplies $C$ by $Q^T$.

- `m` The number of rows in the matrix $C$ ($m \geq 0$).

- `n` The number of columns in $C$ ($n \geq 0$).

- `k` The number of elementary reflectors whose product defines the matrix $Q$.

Constraints:

- $0 \leq k \leq m$ if `side` = 'L';
0 ≤ k ≤ n if side = 'R'.

Arrays:

a and tau are arrays returned by ?gelqf.

The size of a must be:

For side = 'L' and column major layout, max(1, lda*m).
For side = 'R' and column major layout, max(1, lda*n).
For row major layout regardless of side, max(1, lda*k).

The dimension of tau must be at least max(1, k).

c(size max(1, ldc*n) for column major layout and max(1, ldc*m for row major layout) contains the m-by-n matrix C.

Output Parameters

c

Overwritten by the product Q* C, Q^T* C, C* Q, or C* Q^T (as specified by side and trans).

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is unmlq.

?unglq
Generates the complex unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

lapack_int LAPACKE_cunglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zunglq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);

Include Files

• mkl.h

Description
The routine generates the whole or part of \( n \times n \) unitary matrix \( Q \) of the \( LQ \) factorization formed by the routines \texttt{gelqf}. Use this routine after a call to \texttt{cgelqf/zgelqf}.

Usually \( Q \) is determined from the \( LQ \) factorization of an \( p \times n \) matrix \( A \) with \( n < p \). To compute the whole matrix \( Q \), use:

```plaintext
info = LAPACKE_?unglq(matrix_layout, n, n, p, a, lda, tau)
```

To compute the leading \( p \) rows of \( Q \), which form an orthonormal basis in the space spanned by the rows of \( A \), use:

```plaintext
info = LAPACKE_?unglq(matrix_layout, p, n, p, a, lda, tau)
```

To compute the matrix \( Q^k \) of the \( LQ \) factorization of the leading \( k \) rows of \( A \), use:

```plaintext
info = LAPACKE_?unglq(matrix_layout, n, n, k, a, lda, tau)
```

To compute the leading \( k \) rows of \( Q^k \), which form an orthonormal basis in the space spanned by the leading \( k \) rows of \( A \), use:

```plaintext
info = LAPACKE_?ungqr(matrix_layout, k, n, k, a, lda, tau)
```

### Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (\texttt{LAPACK_ROW_MAJOR}) or column major (\texttt{LAPACK_COL_MAJOR}).

- **m**
  - The number of rows of \( Q \) to be computed (\( 0 \leq m \leq n \)).

- **n**
  - The order of the unitary matrix \( Q \) (\( n \geq m \)).

- **k**
  - The number of elementary reflectors whose product defines the matrix \( Q \) (\( 0 \leq k \leq m \)).

- **a, tau**
  - Arrays: \( a \) (size \( \max(1, \text{lda} \times n) \) for column major layout and \( \max(1, \text{lda} \times m) \) for row major layout) and \( tau \) are the arrays returned by \texttt{cgelqf/zgelqf}.
  - The dimension of \( tau \) must be at least \( \max(1, k) \).

- **lda**
  - The leading dimension of \( a \); at least \( \max(1, k) \) for column major layout and \( \max(1, n) \) for row major layout.

### Output Parameters

- **a**
  - Overwritten by \( m \) leading rows of the \( n \times n \) unitary matrix \( Q \).

### Return Values

This function returns a value \( info \).

- If \( info = 0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.

### Application Notes

The computed \( Q \) differs from an exactly unitary matrix by a matrix \( E \) such that \( \| E \|_2 = O(\varepsilon) \times \| A \|_2 \)

where \( \varepsilon \) is the machine precision.

The total number of floating-point operations is approximately \( 16 \times m \times n \times k - 8 \times (m + n) \times k^2 + (16/3) \times k^3 \).

If \( m = k \), the number is approximately \((8/3) \times m^2 \times (3n - m)\).

The real counterpart of this routine is \texttt{orglq}. 

746
?unmlq
Multiplies a complex matrix by the unitary matrix Q of the LQ factorization formed by ?gelqf.

Syntax

```c
lapack_int LAPACKE_cunmlq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```c
lapack_int LAPACKE_zunmlq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a real m-by-n matrix C by Q or Q^H, where Q is the unitary matrix Q of the LQ factorization formed by the routine gelqf.

Depending on the parameters side and trans, the routine can form one of the matrix products Q*C, Q^H*C, C*Q, or C*Q^H (overwriting the result on C).

Input Parameters

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `side`
  Must be either 'L' or 'R'.
  - If side = 'L', Q or Q^H is applied to C from the left.
  - If side = 'R', Q or Q^H is applied to C from the right.

- `trans`
  Must be either 'N' or 'C'.
  - If trans = 'N', the routine multiplies C by Q.
  - If trans = 'C', the routine multiplies C by Q^H.

- `m`
  The number of rows in the matrix C (m ≥ 0).

- `n`
  The number of columns in C (n ≥ 0).

- `k`
  The number of elementary reflectors whose product defines the matrix Q. Constraints:
  - 0 ≤ k ≤ m if side = 'L';
  - 0 ≤ k ≤ n if side = 'R'.

- `a`, `c`, `tau`
  Arrays:
  - `a` and `tau` are arrays returned by ?gelqf.
  - The size of `a` must be:
    - For side = 'L' and column major layout, max(1, lda*m).
    - For side = 'R' and column major layout, max(1, lda*n).
For row major layout regardless of side, max(1, lda*k).
The size of tau must be at least max(1, k).
c(size max(1, ldc*n) for column major layout and max(1, ldc*m for row major layout) contains the m-by-n matrix C.

lda
The leading dimension of a. For column major layout, lda≥ max(1, k). For row major layout, if side = 'L', lda≥ max(1, m), or, if side = 'R', lda≥ max(1, n).

ldc
The leading dimension of c; ldc≥ max(1, m) for column major layout and max(1, n) for row major layout.

Output Parameters

c
Overwritten by the product Q*C, Q^H*C, C*Q, or C*Q^H (as specified by side and trans).

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is ormlq.

?geqlf
Computes the QL factorization of a general m-by-n matrix.

Syntax

lapack_int LAPACKE_sgelqf (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);

lapack_int LAPACKE_dgelqf (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);

lapack_int LAPACKE_cgelqf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgelqf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);

Include Files

• mkl.h

Description

The routine forms the QL factorization of a general m-by-n matrix A (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix Q explicitly. Instead, Q is represented as a product of min(m, n) elementary reflectors. Routines are provided to work with Q in this representation.
NOTE
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

m
The number of rows in the matrix \( A \) (\( m \geq 0 \)).

n
The number of columns in \( A \) (\( n \geq 0 \)).

a
Array \( a \) of size \( \max(1, lda*n) \) for column major layout and \( \max(1, lda*m) \) for row major layout contains the matrix \( A \).

lda
The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

Output Parameters

a
Overwritten on exit by the factorization data as follows:
if \( m \geq n \), the lower triangle of the subarray \( a(m-n+1:m, 1:n) \) contains the \( n \)-by-\( n \) lower triangular matrix \( L \); if \( m \leq n \), the elements on and below the \( (n-m) \)-th superdiagonal contain the \( m \)-by-\( n \) lower trapezoidal matrix \( L \); in both cases, the remaining elements, with the array \( tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors.

tau
Array, size at least \( \max(1, \min(m, n)) \). Contains scalar factors of the elementary reflectors for the matrix \( Q \) (see Orthogonal Factorizations).

Return Values
This function returns a value \( info \).
If \( info=0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

Application Notes
Related routines include:

- orgql to generate matrix \( Q \) (for real matrices);
- ungql to generate matrix \( Q \) (for complex matrices);
- ormql to apply matrix \( Q \) (for real matrices);
- unmql to apply matrix \( Q \) (for complex matrices).

See Also
mkl_progress

Matrix Storage Schemes
?orgql
Generates the real matrix Q of the QL factorization formed by ?geqlf.

Syntax

lapack_int LAPACKE_sorgql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);

lapack_int LAPACKE_dorgql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);

Include Files

• mkl.h

Description

The routine generates an m-by-n real matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors \( H(i) \) of order m: \( Q = H(k) \cdots H(2)H(1) \) as returned by the routines ?geqlf. Use this routine after a call to ?sgeqlf/dgeqlf.

Input Parameters

- **matrix_layout**
  Specifications whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **m**
  The number of rows of the matrix Q (\( m \geq 0 \)).

- **n**
  The number of columns of the matrix Q (\( m \geq n \geq 0 \)).

- **k**
  The number of elementary reflectors whose product defines the matrix Q (\( n \geq k \geq 0 \)).

- **a, tau**
  Arrays: a (size max(1, lda*n) for column major layout and max(1, lda*m) for row major layout), tau.
  On entry, the (n - k + i)th column of a must contain the vector which defines the elementary reflector \( H(i) \), for \( i = 1,2,\ldots,k \), as returned by ?sgeqlf/dgeqlf in the last k columns of its array argument a; tau[i - 1] must contain the scalar factor of the elementary reflector \( H(i) \), as returned by ?sgeqlf/dgeqlf;
  The size of tau must be at least max(1, k).

- **lda**
  The leading dimension of a; at least max(1, m) for column major layout and max(1, n) for row major layout.

Output Parameters

- **a**
  Overwritten by the last n columns of the m-by-m orthogonal matrix Q.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
Application Notes
The complex counterpart of this routine is ungql.

?ungql
Generates the complex matrix Q of the QL factorization formed by ?geqlf.

Syntax

```c
lapack_int LAPACKE_cungql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
                         lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);
```

```c
lapack_int LAPACKE_zungql (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
                         lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine generates an m-by-n complex matrix Q with orthonormal columns, which is defined as the last n columns of a product of k elementary reflectors \( H(i) \) of order m: \( Q = H(k) \cdots H(2)H(1) \) as returned by the routines geqlf/zgeqlf. Use this routine after a call to cgeqlf/zgeqlf.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **m**
  The number of rows of the matrix Q \((m\geq 0)\).

- **n**
  The number of columns of the matrix Q \((m\geq n\geq 0)\).

- **k**
  The number of elementary reflectors whose product defines the matrix Q \((n\geq k \geq 0)\).

- **a, tau**
  Arrays: a (size max(1, lda*n) for column major layout and max(1, lda*m) for row major layout), tau.
  On entry, the \((n - k + i)\)th column of a must contain the vector which defines the elementary reflector \( H(i) \), for i = 1,2,...,k, as returned by cgeqlf/zgeqlf in the last k columns of its array argument a;
  \( tau[i - 1] \) must contain the scalar factor of the elementary reflector \( H(i) \), as returned by cgeqlf/zgeqlf;
  The size of \( tau \) must be at least max(1, k).

- **lda**
  The leading dimension of a; at least max(1, m) for column major layout and max(1, n) for row major layout.

Output Parameters

- **a**
  Overwritten by the last n columns of the m-by-m unitary matrix Q.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes
The real counterpart of this routine is orgql.

.ormql
Multiplying a real matrix by the orthogonal matrix Q of the QL factorization formed by ?geqlf.

Syntax

```c
lapack_int LAPACKE_sormql (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c, lapack_int ldc);
```

Include Files
- mkl.h

Description
The routine multiplies a real m-by-n matrix C by Q or QT, where Q is the orthogonal matrix Q of the QL factorization formed by the routine geqlf.

Depending on the parameters side and trans, the routine ormlq can form one of the matrix products Q*C, Q^T*C, C*Q, or C*Q^T (overwriting the result over C).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **side**
  Must be either 'L' or 'R'.
  - If side = 'L', Q or QT is applied to C from the left.
  - If side = 'R', Q or QT is applied to C from the right.

- **trans**
  Must be either 'N' or 'T'.
  - If trans = 'N', the routine multiplies C by Q.
  - If trans = 'T', the routine multiplies C by QT.

- **m**
  The number of rows in the matrix C (m ≥ 0).

- **n**
  The number of columns in C (n ≥ 0).

- **k**
  The number of elementary reflectors whose product defines the matrix Q.
  Constraints:
  - 0 ≤ k ≤ m if side = 'L';
  - 0 ≤ k ≤ n if side = 'R'.

752
Arrays: $a$, $\tau$, $c$.

The size of $a$ must be:

For column major layout regardless of $\text{side}$, $\max(1, \text{lda} \times k)$.

For $\text{side} = \text{'L'}$ and row major layout, $\max(1, \text{lda} \times m)$.

For $\text{side} = \text{'R'}$ and row major layout, $\max(1, \text{lda} \times n)$.

On entry, the $i$th column of $a$ must contain the vector which defines the elementary reflector $H_i$, for $i = 1, 2, ..., k$, as returned by \texttt{sgeqlf/dgeqlf} in the last $k$ columns of its array argument $a$.

$\tau[i-1]$ must contain the scalar factor of the elementary reflector $H_i$, as returned by \texttt{sgeqlf/dgeqlf}.

The size of $\tau$ must be at least $\max(1, k)$.

$c$(size $\max(1, \text{ldc} \times n)$ for column major layout and $\max(1, \text{ldc} \times m)$ for row major layout) contains the $m$-by-$n$ matrix $C$.

$\text{lda}$

The leading dimension of $a$;

if $\text{side} = \text{'L'}$, $\text{lda} \geq \max(1, m)$ for column major layout and $\max(1, k)$ for row major layout;

if $\text{side} = \text{'R'}$, $\text{lda} \geq \max(1, n)$ for column major layout and $\max(1, k)$ for row major layout.

$\text{ldc}$

The leading dimension of $c$; $\text{ldc} \geq \max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

Output Parameters

$c$

Overwritten by the product $Q^*C$, $Q^*C$, $C^*Q$, or $C^*Q^*$ (as specified by $\text{side}$ and $\text{trans}$).

Return Values

This function returns a value $\text{info}$.

If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is \texttt{unmql}.

\texttt{?unmql}

\textit{Multiplies a complex matrix by the unitary matrix $Q$ of the QL factorization formed by \texttt{?geqlf}.}

Syntax

\begin{verbatim}
lapack_int LAPACKE_cunmql (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmql (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
\end{verbatim}
Include Files

• mkl.h

Description

The routine multiplies a complex $m$-by-$n$ matrix $C$ by $Q$ or $Q^H$, where $Q$ is the unitary matrix $Q$ of the $QL$ factorization formed by the routine geqlf.

Depending on the parameters $side$ and $trans$, the routine unmql can form one of the matrix products $Q*C$, $Q^H*C$, $C*Q$, or $C*Q^H$ (overwriting the result over $C$).

Input Parameters

(matrix_layout)

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

(side)

Must be either 'L' or 'R'.

If $side$ = 'L', $Q$ or $Q^H$ is applied to $C$ from the left.

If $side$ = 'R', $Q$ or $Q^H$ is applied to $C$ from the right.

(trans)

Must be either 'N' or 'C'.

If $trans$ = 'N', the routine multiplies $C$ by $Q$.

If $trans$ = 'C', the routine multiplies $C$ by $Q^H$.

$m$

The number of rows in the matrix $C$ ($m \geq 0$).

$n$

The number of columns in $C$ ($n \geq 0$).

$k$

The number of elementary reflectors whose product defines the matrix $Q$.

Constraints:

$0 \leq k \leq m$ if $side$ = 'L';

$0 \leq k \leq n$ if $side$ = 'R'.

(a, tau, c)

Arrays: $a$, $tau$, $c$.

The size of $a$ must be:

For column major layout regardless of $side$, max(1, $lda*k$).

For $side$ = 'L' and row major layout, max(1, $lda*m$).

For $side$ = 'R' and row major layout, max(1, $lda*n$).

On entry, the $i$-th column of $a$ must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, ..., k$, as returned by cgeqlf/zgeqlf in the last $k$ columns of its array argument $a$.

$tau[i - 1]$ must contain the scalar factor of the elementary reflector $H(i)$, as returned by cgeqlf/zgeqlf.

The size of $tau$ must be at least max(1, $k$).

$c$(size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$ for row major layout) contains the $m$-by-$n$ matrix $C$.

$lda$

The leading dimension of $a$. 

If \( \text{side} = 'L' \), \( \text{lda} \geq \max(1, \ m) \) for column major layout and \( \max(1, \ k) \) for row major layout.

If \( \text{side} = 'R' \), \( \text{lda} \geq \max(1, \ n) \) for column major layout and \( \max(1, \ k) \) for row major layout.

\( \text{ldc} \)

The leading dimension of \( c \); \( \text{ldc} \geq \max(1, \ m) \) for column major layout and \( \max(1, \ n) \) for row major layout.

**Output Parameters**

\( c \)

Overwritten by the product \( Q*C, Q^H*C, C*Q \), or \( C*Q^H \) (as specified by \( \text{side} \) and \( \text{trans} \)).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The real counterpart of this routine is \texttt{ormql}.

\texttt{?gerqf}

*Computes the RQ factorization of a general \( m \)-by-\( n \) matrix.*

**Syntax**

\[
\begin{align*}
\text{lapack_int LAPACKE_sgerqf} & \quad (\text{int } \text{matrix_layout}, \text{lapack_int } m, \text{lapack_int } n, \text{float* } a, \\
& \quad \text{lapack_int } \text{lda}, \text{float* } \text{tau}); \\
\text{lapack_int LAPACKE_dgerqf} & \quad (\text{int } \text{matrix_layout}, \text{lapack_int } m, \text{lapack_int } n, \text{double* } a, \\
& \quad \text{lapack_int } \text{lda}, \text{double* } \text{tau}); \\
\text{lapack_int LAPACKE_cgerqf} & \quad (\text{int } \text{matrix_layout}, \text{lapack_int } m, \text{lapack_int } n, \\
& \quad \text{lapack_complex_float* } a, \text{lapack_int } \text{lda}, \text{lapack_complex_float* } \text{tau}); \\
\text{lapack_int LAPACKE_zgerqf} & \quad (\text{int } \text{matrix_layout}, \text{lapack_int } m, \text{lapack_int } n, \\
& \quad \text{lapack_complex_double* } a, \text{lapack_int } \text{lda}, \text{lapack_complex_double* } \text{tau});
\end{align*}
\]

**Include Files**

- \texttt{mkl.h}

**Description**

The routine forms the \( RQ \) factorization of a general \( m \)-by-\( n \) matrix \( A \) (see Orthogonal Factorizations). No pivoting is performed.

The routine does not form the matrix \( Q \) explicitly. Instead, \( Q \) is represented as a product of \( \min(m, n) \) \textit{elementary reflectors}. Routines are provided to work with \( Q \) in this representation.

\textbf{NOTE}

This routine supports the Progress Routine feature. See Progress Function section for details.
Input Parameters

\textit{matrix\_layout} Specifies whether matrix storage layout is row major (LAPACK\_ROW\_MAJOR) or column major (LAPACK\_COL\_MAJOR).

\textit{m} The number of rows in the matrix \( A \) (\( m \geq 0 \)).

\textit{n} The number of columns in \( A \) (\( n \geq 0 \)).

\textit{a} Array \( a \) of size \( \max(1, lda \times n) \) for column major layout and \( \max(1, lda \times m) \) for row major layout contains the \( m \)-by-\( n \) matrix \( A \).

\textit{lda} The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

Output Parameters

\textit{a} Overwritten on exit by the factorization data as follows:

- if \( m \leq n \), the upper triangle of the subarray \( a(1:m, n-m+1:n) \) contains the \( m \)-by-\( m \) upper triangular matrix \( R \);
- if \( m \geq n \), the elements on and above the \((m-n)\)th subdiagonal contain the \( m \)-by-\( n \) upper trapezoidal matrix \( R \);

in both cases, the remaining elements, with the array \( \tau \), represent the orthogonal/unitary matrix \( Q \) as a product of \( \min(m, n) \) elementary reflectors.

\textit{tau} Array, size at least \( \max(1, \min(m, n)) \). (See Orthogonal Factorizations.)

Contains scalar factors of the elementary reflectors for the matrix \( Q \).

Return Values

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

Related routines include:

- \textit{orgqr} to generate matrix \( Q \) (for real matrices);
- \textit{ungrq} to generate matrix \( Q \) (for complex matrices);
- \textit{ormrq} to apply matrix \( Q \) (for real matrices);
- \textit{unmrq} to apply matrix \( Q \) (for complex matrices).

See Also

\textit{mkl\_progress}

Matrix Storage Schemes
?orgrq
Generates the real matrix Q of the RQ factorization
formed by ?gerqf.

Syntax

```c
lapack_int LAPACKE_sorgrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
float* a, lapack_int lda, const float* tau);

lapack_int LAPACKE_dorgrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k,
double* a, lapack_int lda, const double* tau);
```

Include Files

- mkl.h

Description

The routine generates an m-by-n real matrix with orthonormal rows, which is defined as the last m rows of a
product of k elementary reflectors H(i) of order n: Q = H(1)* H(2)*...*H(k) as returned by the routines
gerqf. Use this routine after a call to sgerqf/dgerqf.

Input Parameters

- `matrix_layout`: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
  or column major (LAPACK_COL_MAJOR).
- `m`: The number of rows of the matrix Q (m ≥ 0).
- `n`: The number of columns of the matrix Q (n ≥ m).
- `k`: The number of elementary reflectors whose product defines the matrix Q
  (m ≥ k ≥ 0).
- `a, tau`: Arrays: a(size max(1, lda*n) for column major layout and max(1, lda*m)
  for row major layout), tau.

  On entry, the (m - k + i)-th row of a must contain the vector which defines
  the elementary reflector H(i), for i = 1,2,...,k, as returned by sgerqf/
dgerqf in the last k rows of its array argument a;

  tau[i - 1] must contain the scalar factor of the elementary reflector H(i),
  as returned by sgerqf/dgerqf;

  The size of tau must be at least max(1, k).

- `lda`: The leading dimension of a; at least max(1, m)for column major layout and
  max(1, n) for row major layout.

Output Parameters

- `a`: Overwritten by the last m rows of the n-by-n orthogonal matrix Q.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.
Application Notes

The complex counterpart of this routine is `ungrq`.

`?ungrq`  
*Generates the complex matrix Q of the RQ factorization formed by ?gerqf.*

Syntax

```c
lapack_int LAPACKE_cungrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, 
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zungrq (int matrix_layout, lapack_int m, lapack_int n, lapack_int k, 
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- `mkl.h`

Description

The routine generates an \( m \)-by-\( n \) complex matrix with orthonormal rows, which is defined as the last \( m \) rows of a product of \( k \) elementary reflectors \( H(i) \) of order \( n \): 

\[
Q = \prod_{i=1}^{k} H(i)^H
\]

as returned by the routines `gerqf`. Use this routine after a call to `cgerqf/zgerqf`.

Input Parameters

- `matrix_layout`  
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).
- `m`  
  The number of rows of the matrix \( Q \) (\( m \geq 0 \)).
- `n`  
  The number of columns of the matrix \( Q \) (\( n \geq m \)).
- `k`  
  The number of elementary reflectors whose product defines the matrix \( Q \) (\( m \geq k \geq 0 \)).
- `a`, `tau`  
  Arrays: \( a \) (size max(1, \( lda \times n \)) for column major layout and max(1, \( lda \times m \)) for row major layout), \( tau \).
  On entry, the \( (m - k + i) \)th row of \( a \) must contain the vector which defines the elementary reflector \( H(i) \), for \( i = 1,2,...,k \), as returned by `cgerqf/zgerqf` in the last \( k \) rows of its array argument \( a \);
  \( tau[i - 1] \) must contain the scalar factor of the elementary reflector \( H(i) \), as returned by `cgerqf/zgerqf`;
  The size of \( tau \) must be at least max(1, \( k \)).
- `lda`  
  The leading dimension of \( a \); at least max(1, \( m \)) for column major layout and max(1, \( n \)) for row major layout.

Output Parameters

- `a`  
  Overwritten by the \( m \) last rows of the \( n \)-by-\( n \) unitary matrix \( Q \).

Return Values

This function returns a value \( info \).
If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

The real counterpart of this routine is orgrq.

**ormrq**

*Multiplying a real matrix by the orthogonal matrix $Q$ of the $RQ$ factorization formed by ?gerqf.*

**Syntax**

```c
lapack_int LAPACKE_sormrq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* tau, float* c, lapack_int ldc);
```

```c
lapack_int LAPACKE_dormrq (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, const double* a, lapack_int lda, const double* tau, double* c, lapack_int ldc);
```

**Include Files**

- mkl.h

**Description**

The routine multiplies a real $m$-by-$n$ matrix $C$ by $Q$ or $Q^T$, where $Q$ is the real orthogonal matrix defined as a product of $k$ elementary reflectors $H_i: Q = H_1H_2 \ldots H_k$ as returned by the $RQ$ factorization routine ?gerqf.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^*C$, $Q^T*C$, $C*Q$, or $C*Q^T$ (overwriting the result over $C$).

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `side` Must be either 'L' or 'R'.
  - If `side` = 'L', $Q$ or $Q^T$ is applied to $C$ from the left.
  - If `side` = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.
- `trans` Must be either 'N' or 'T'.
  - If `trans` = 'N', the routine multiplies $C$ by $Q$.
  - If `trans` = 'T', the routine multiplies $C$ by $Q^T$.
- `m` The number of rows in the matrix $C$ ($m \geq 0$).
- `n` The number of columns in $C$ ($n \geq 0$).
- `k` The number of elementary reflectors whose product defines the matrix $Q$.

Constraints:
- $0 \leq k \leq m$, if `side` = 'L';
- $0 \leq k \leq n$, if `side` = 'R'.

LAPACK Routines
Arrays: $a$ (size for $side$ = 'L': max(1, $lda*m$) for column major layout and max(1, $lda*k$) for row major layout; for $side$ = 'R': max(1, $lda*n$) for column major layout and max(1, $lda*k$) for row major layout), $tau$, $c$ (size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$) for row major layout).

On entry, the $i$th row of $a$ must contain the vector which defines the elementary reflector $H_i$, for $i = 1,2,...,k$, as returned by sgerqf/dgerqf in the last $k$ rows of its array argument $a$.

$tau[i - 1]$ must contain the scalar factor of the elementary reflector $H_i$, as returned by sgerqf/dgerqf.

The size of $tau$ must be at least max(1, $k$).

$c$ contains the $m$-by-$n$ matrix $C$.

$lda$ The leading dimension of $a$; $lda \geq$ max(1, $k$) for column major layout. For row major layout, $lda \geq$ max(1, $m$) if $side$ = 'L', and $lda \geq$ max(1, $n$) if $side$ = 'R'.

$ldc$ The leading dimension of $c$; $ldc \geq$ max(1, $m$) for column major layout and max(1, $n$) for row major layout.

Output Parameters

$c$ Overwritten by the product $Q\cdot C$, $Q^T\cdot C$, $C\cdot Q$, or $C\cdot Q^T$ (as specified by $side$ and $trans$).

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is unmrq.

?unmrq

Multiplies a complex matrix by the unitary matrix $Q$ of the RQ factorization formed by ?gerqf.

Syntax

```c
lapack_int LAPACKE_cunmrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda,
const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmrq (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h
Description

The routine multiplies a complex $m$-by-$n$ matrix $C$ by $Q$ or $Q^H$, where $Q$ is the complex unitary matrix defined as a product of $k$ elementary reflectors $H(i)$ of order $n$: $Q = H(1)^* H(2)^* \ldots H(k)^*$ as returned by the RQ factorization routine \texttt{gerqf}.

Depending on the parameters \texttt{side} and \texttt{trans}, the routine can form one of the matrix products $Q*C$, $Q^H*C$, $C*Q$, or $C*Q^H$ (overwriting the result over $C$).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (\texttt{LAPACK Row Major}) or column major (\texttt{LAPACK Col Major}).

- **side**
  Must be either 'L' or 'R'.
  - If \texttt{side} = 'L', $Q$ or $Q^H$ is applied to $C$ from the left.
  - If \texttt{side} = 'R', $Q$ or $Q^H$ is applied to $C$ from the right.

- **trans**
  Must be either 'N' or 'C'.
  - If \texttt{trans} = 'N', the routine multiplies $C$ by $Q$.
  - If \texttt{trans} = 'C', the routine multiplies $C$ by $Q^H$.

- **m**
  The number of rows in the matrix $C$ ($m \geq 0$).

- **n**
  The number of columns in $C$ ($n \geq 0$).

- **k**
  The number of elementary reflectors whose product defines the matrix $Q$.
  Constraints:
  - $0 \leq k \leq m$, if \texttt{side} = 'L';
  - $0 \leq k \leq n$, if \texttt{side} = 'R'.

- **a**, **tau**, **c**
  Arrays: $a$ (size for \texttt{side} = 'L': max(1, $lda*m$) for column major layout and max(1, $lda*k$) for row major layout; for \texttt{side} = 'R': max(1, $lda*n$) for column major layout and max(1, $lda*k$) for row major layout), \texttt{tau}, \texttt{c} (size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$) for row major layout).

  On entry, the $i$th row of $a$ must contain the vector which defines the elementary reflector $H(i)$, for $i = 1, 2, \ldots, k$, as returned by \texttt{cgerqf/zgerqf} in the last $k$ rows of its array argument $a$.

  \texttt{tau[i - 1]} must contain the scalar factor of the elementary reflector $H(i)$, as returned by \texttt{cgerqf/zgerqf}.

  The size of \texttt{tau} must be at least max(1, $k$).

- **lda**
  The leading dimension of $a$; $lda \geq$ max(1, $k$) for column major layout. For row major layout, $lda \geq$ max(1, $m$) if \texttt{side} = 'L', and $lda \geq$ max(1, $n$) if \texttt{side} = 'R'.

- **ldc**
  The leading dimension of $c$; $ldc \geq$ max(1, $m$) for column major layout and max(1, $n$) for row major layout.
Output Parameters

c
Overwritten by the product \( Q^*C, Q^{H}*C, C^*Q, \) or \( C^*Q^H \) (as specified by \textit{side} and \textit{trans}).

Return Values

This function returns a value \textit{info}.

If \( \textit{info}=0 \), the execution is successful.

If \( \textit{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is \texttt{ormrq}.

\(?tzrzf\)

\textit{Reduces the upper trapezoidal matrix \textit{A} to upper triangular form.}

Syntax

\begin{verbatim}
lapack_int LAPACKE_stzrzf (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);
lapack_int LAPACKE_dtzrzf (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);
lapack_int LAPACKE_ctzrzf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
lapack_int LAPACKE_ztzrzf (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
\end{verbatim}

Include Files

- \texttt{mkl.h}

Description

The routine reduces the \( m \)-by-\( n \) \((m \leq n)\) real/complex upper trapezoidal matrix \( A \) to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix \( A = [A1 \hspace{1em} A2] = [A1:m, 1:m, A1:m, m+1:n] \) is factored as

\[ A = [R0]^*Z, \]

where \( Z \) is an \( n \)-by-\( n \) orthogonal/unitary matrix, \( R \) is an \( m \)-by-\( m \) upper triangular matrix, and \( 0 \) is the \( m \)-by-\( (n-m) \) zero matrix.

The \(?tzrzf\) routine replaces the deprecated \(?tzrqf\) routine.

Input Parameters

\begin{verbatim}
matrix_layout
\end{verbatim}

Specifies whether matrix storage layout is row major (\texttt{LAPACK_ROW_MAJOR}) or column major (\texttt{LAPACK_COL_MAJOR}).

\begin{verbatim}
m
\end{verbatim}
The number of rows in the matrix \( A \) \((n \geq 0)\).

\begin{verbatim}
n
\end{verbatim}
The number of columns in \( A \) \((n \geq m)\).
\(a\) is an array of size \(\max(1, \text{lda} \times n)\) for column major layout and \(\max(1, \text{lda} \times m)\) for row major layout.

The leading \(m\)-by-\(n\) upper trapezoidal part of the array \(a\) contains the matrix \(A\) to be factorized.

\(\text{lda}\)

The leading dimension of \(a\); at least \(\max(1, m)\) for column major layout and \(\max(1, n)\) for row major layout.

**Output Parameters**

\(a\)

Overwritten on exit by the factorization data as follows:

the leading \(m\)-by-\(m\) upper triangular part of \(a\) contains the upper triangular matrix \(R\), and elements \(m + 1\) to \(n\) of the first \(m\) rows of \(a\), with the array \(\text{tau}\), represent the orthogonal matrix \(Z\) as a product of \(m\) elementary reflectors.

\(\text{tau}\)

Array, size at least \(\max(1, m)\). Contains scalar factors of the elementary reflectors for the matrix \(Z\).

**Return Values**

This function returns a value \(\text{info}\).

If \(\text{info} = 0\), the execution is successful.

If \(\text{info} = -i\), the \(i\)-th parameter had an illegal value.

**Application Notes**

The factorization is obtained by Householder's method. The \(k\)-th transformation matrix, \(Z(k)\), which is used to introduce zeros into the \((m - k + 1)\)-th row of \(A\), is given in the form

\[
Z(k) = \begin{bmatrix}
I & 0 \\
0 & T(k)
\end{bmatrix}
\]

where for real flavors

\[
T(k) = I - t \alpha k^* \alpha(k)^T, \quad \alpha(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}
\]

and for complex flavors

\[
T(k) = I - t \alpha k^* \alpha(k)^H, \quad \alpha(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}
\]

\(\text{tau}\) is a scalar and \(z(k)\) is an \(l\)-element vector. \(\text{tau}\) and \(z(k)\) are chosen to annihilate the elements of the \(k\)-th row of \(A_2\).

The scalar \(\text{tau}\) is returned in the \(k\)-th element of \(\text{tau}\) and the vector \(u(k)\) in the \(k\)-th row of \(A\), such that the elements of \(z(k)\) are stored in the last \(m - n\) elements of the \(k\)-th row of array \(a\).

The elements of \(R\) are returned in the upper triangular part of \(A\).
The matrix $Z$ is given by
\[ Z = Z(1)*Z(2)*...*Z(m). \]

Related routines include:
- `ormrz` to apply matrix $Q$ (for real matrices)
- `unmrz` to apply matrix $Q$ (for complex matrices).

**?ormrz**

*Multiplies a real matrix by the orthogonal matrix defined from the factorization formed by `?tzrzf`.*

**Syntax**

```c
lapack_int LAPACKE_sormrz (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const float* a, lapack_int lda, const float* tau, float* c, lapack_int ldc);

lapack_int LAPACKE_dormrz (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const double* a, lapack_int lda, const double* tau, double* c, lapack_int ldc);
```

**Include Files**

- `mkl.h`

**Description**

The `?ormrz` routine multiplies a real $m$-by-$n$ matrix $C$ by $Q$ or $Q^T$, where $Q$ is the real orthogonal matrix defined as a product of $k$ elementary reflectors $H(i)$ of order $n$: $Q = H(1)*H(2)*...*H(k)$ as returned by the factorization routine `?tzrzf`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q*C$, $Q^T*C$, $C*Q$, or $C*Q^T$ (overwriting the result over $C$).

The matrix $Q$ is of order $m$ if `side` = 'L' and of order $n$ if `side` = 'R'.

The `?ormrz` routine replaces the deprecated `?latzm` routine.

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `side` Must be either 'L' or 'R'.
  - If `side` = 'L', $Q$ or $Q^T$ is applied to $C$ from the left.
  - If `side` = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.
- `trans` Must be either 'N' or 'T'.
  - If `trans` = 'N', the routine multiplies $C$ by $Q$.
  - If `trans` = 'T', the routine multiplies $C$ by $Q^T$.
- `m` The number of rows in the matrix $C$ ($m \geq 0$).
- `n` The number of columns in $C$ ($n \geq 0$).
The number of elementary reflectors whose product defines the matrix \( Q \).

Constraints:

- \( 0 \leq k \leq m \), if \( \text{side} = 'L' \);
- \( 0 \leq k \leq n \), if \( \text{side} = 'R' \).

The number of columns of the matrix \( A \) containing the meaningful part of
the Householder reflectors. Constraints:

- \( 0 \leq l \leq m \), if \( \text{side} = 'L' \);
- \( 0 \leq l \leq n \), if \( \text{side} = 'R' \).

Arrays: \( a \) (size for \( \text{side} = 'L' \): \( \max(1, \text{lda}*m) \) for column major layout and
\( \max(1, \text{lda}*k) \) for row major layout; for \( \text{side} = 'R' \): \( \max(1, \text{lda}*b) \) for column major layout and \( \max(1, \text{lda}*k) \) for row major layout), \( \tau \), \( c \) (size \( \max(1, \text{ldc}*n) \) for column major layout and \( \max(1, \text{ldc}*m) \) for row major layout).

On entry, the \( i \)th row of \( a \) must contain the vector which defines the
elementary reflector \( H(i) \), for \( i = 1, 2, \ldots, k \), as returned by \( \text{stzrzf/dtzrzf} \)
in the last \( k \) rows of its array argument \( a \).

\( \tau[i - 1] \) must contain the scalar factor of the elementary reflector \( H(i) \),
as returned by \( \text{stzrzf/dtzrzf} \).

The size of \( \tau \) must be at least \( \max(1, k) \).

\( c \) contains the \( m \)-by-\( n \) matrix \( C \).

The leading dimension of \( a \); \( \text{lda} \geq \max(1, k) \) for column major layout. For row
major layout, \( \text{lda} \geq \max(1, m) \) if \( \text{side} = 'L' \), and \( \text{lda} \geq \max(1, n) \) if \( \text{side} = 'R' \).

The leading dimension of \( c \); \( \text{ldc} \geq \max(1, m) \) for column major layout and
\( \max(1, n) \) for row major layout.

Output Parameters

\( c \)

Overwritten by the product \( Q^*C \), \( Q^T*C \), \( C^*Q \), or \( C^TQ \) (as specified by \( \text{side} \)
and \( \text{trans} \)).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The complex counterpart of this routine is \( \text{unmrz} \).

\( \text{unmrz} \)

Multiplies a complex matrix by the unitary matrix
defined from the factorization formed by \( \text{stzrzf} \).
Syntax

```c
lapack_int LAPACKE_cunmrz (int matrix_layout, char side, char trans, lapack_int m,
    lapack_int n, lapack_int k, lapack_int l, const lapack_complex_float* a, lapack_int lda,
    const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```c
lapack_int LAPACKE_zunmrz (int matrix_layout, char side, char trans, lapack_int m,
    lapack_int n, lapack_int k, lapack_int l, const lapack_complex_double* a, lapack_int lda,
    const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

The routine multiplies a complex \( m \)-by-\( n \) matrix \( C \) by \( Q \) or \( Q^H \), where \( Q \) is the unitary matrix defined as a product of \( k \) elementary reflectors \( H(i) \):

\[
Q = H(1)^H \cdot H(2)^H \cdot \ldots \cdot H(k)^H \quad \text{as returned by the factorization routine tzrzf}.
\]

Depending on the parameters \( side \) and \( trans \), the routine can form one of the matrix products \( Q^*C \), \( Q^H C \), \( C^*Q \), or \( C^*Q^H \) (overwriting the result over \( C \)).

The matrix \( Q \) is of order \( m \) if \( side = 'L' \) and of order \( n \) if \( side = 'R' \).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **side**
  Must be either 'L' or 'R'.
  - If \( side = 'L' \), \( Q \) or \( Q^H \) is applied to \( C \) from the left.
  - If \( side = 'R' \), \( Q \) or \( Q^H \) is applied to \( C \) from the right.

- **trans**
  Must be either 'N' or 'C'.
  - If \( trans = 'N' \), the routine multiplies \( C \) by \( Q \).
  - If \( trans = 'C' \), the routine multiplies \( C \) by \( Q^H \).

- **m**
  The number of rows in the matrix \( C \) (\( m \geq 0 \)).

- **n**
  The number of columns in \( C \) (\( n \geq 0 \)).

- **k**
  The number of elementary reflectors whose product defines the matrix \( Q \).
  Constraints:
  - \( 0 \leq k \leq m \), if \( side = 'L' \);  
  - \( 0 \leq k \leq n \), if \( side = 'R' \).

- **l**
  The number of columns of the matrix \( A \) containing the meaningful part of the Householder reflectors. Constraints:
  - \( 0 \leq l \leq m \), if \( side = 'L' \);  
  - \( 0 \leq l \leq n \), if \( side = 'R' \).
Arrays: \( a \) (size for \( \text{side} = 'L' \): \( \max(1, lda*m) \) for column major layout and \( \max(1, lda*k) \) for row major layout; for \( \text{side} = 'R' \): \( \max(1, lda*b) \) for column major layout and \( \max(1, lda*k) \) for row major layout), \( \tau_a, c \) (size \( \max(1, ldc*n) \) for column major layout and \( \max(1, ldc*m) \) for row major layout).

On entry, the \( i \)th row of \( a \) must contain the vector which defines the elementary reflector \( H(i) \), for \( i = 1,2,...,k \), as returned by \texttt{ctzrzf/ztzrzf} in the last \( k \) rows of its array argument \( a \).

\( \tau_a[i - 1] \) must contain the scalar factor of the elementary reflector \( H(i) \), as returned by \texttt{ctzrzf/ztzrzf}.

The size of \( \tau_a \) must be at least \( \max(1, k) \).

\( c \) contains the \( m \)-by-\( n \) matrix \( C \).

\( \ell_a \) The leading dimension of \( a \); \( \ell_a \geq \max(1, k) \) for column major layout. For row major layout, \( \ell_a \geq \max(1, m) \) if \( \text{side} = 'L' \), and \( \ell_a \geq \max(1, n) \) if \( \text{side} = 'R' \).

\( \ell_c \) The leading dimension of \( c \); \( \ell_c \geq \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

Output Parameters

\( c \) Overwritten by the product \( Q*C, Q^H*C, C*Q \), or \( C*Q^H \) (as specified by \text{side} and \text{trans}).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The real counterpart of this routine is \texttt{ormrz}.

\texttt{?ggqrf}

Computes the generalized QR factorization of two matrices.

Syntax

\begin{verbatim}
lapack_int LAPACKE_sggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p, float* a, lapack_int lda, float* taua, float* b, lapack_int ldb, float* taub);

lapack_int LAPACKE_dggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p, double* a, lapack_int lda, double* taua, double* b, lapack_int ldb, double* taub);

lapack_int LAPACKE_cggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p, lapack_complex_float* a, lapack_int lda, lapack_complex_float* taua, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* taub);

lapack_int LAPACKE_zggqrf (int matrix_layout, lapack_int n, lapack_int m, lapack_int p, lapack_complex_double* a, lapack_int lda, lapack_complex_double* taua, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* taub);
\end{verbatim}
Include Files

- mkl.h

Description

The routine forms the generalized QR factorization of an \( n \)-by-\( m \) matrix \( A \) and an \( n \)-by-\( p \) matrix \( B \) as \( A = Q^*R \), \( B = Q^*T^*Z \), where \( Q \) is an \( n \)-by-\( n \) orthogonal/unitary matrix, \( Z \) is a \( p \)-by-\( p \) orthogonal/unitary matrix, and \( R \) and \( T \) assume one of the forms:

\[
R = \begin{cases} \\
\begin{pmatrix} R_{11} \\ 0 \end{pmatrix} & , \quad \text{if } n \geq m \\
\end{cases} \\
\begin{pmatrix} n - m \\ m - n \end{pmatrix} \\
\end{cases}
\]

or

\[
R = \begin{cases} \\
\begin{pmatrix} R_{11} & R_{12} \\ R_{11} & R_{12} \end{pmatrix} & , \quad \text{if } n < m \\
\end{cases} \\
\end{cases}
\]

where \( R_{11} \) is upper triangular, and

\[
T = \begin{cases} \\
\begin{pmatrix} p - n & n \\ n & T_{11} \end{pmatrix} & , \quad \text{if } n \leq p, \\
\end{cases} \\
\begin{pmatrix} p & \end{pmatrix} \\
\begin{pmatrix} p \\ T_{21} \end{pmatrix} \\
\end{cases}
\]

where \( T_{12} \) or \( T_{21} \) is a \( p \)-by-\( p \) upper triangular matrix.

In particular, if \( B \) is square and nonsingular, the GQR factorization of \( A \) and \( B \) implicitly gives the QR factorization of \( B^{-1}A \) as:

\[
B^{-1}A = Z^* (T^{-1}R) \quad \text{(for real flavors)} \quad \text{or} \quad B^{-1}A = Z^H (T^{-1}R) \quad \text{(for complex flavors)}.
\]

Input Parameters

- \textit{matrix\_layout} Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- \textit{n} The number of rows of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

- \textit{m} The number of columns in \( A \) (\( m \geq 0 \)).

- \textit{p} The number of columns in \( B \) (\( p \geq 0 \)).

- \( a, b \)

  Array \( a \) of size max(1, \( lda \times m \)) for column major layout and max(1, \( lda \times n \)) for row major layout contains the matrix \( A \).

  Array \( b \) of size max(1, \( ldb \times p \)) for column major layout and max(1, \( ldb \times n \)) for row major layout contains the matrix \( B \).
The leading dimension of \(a\); at least max(1, \(n\)) for column major layout and at least max(1, \(m\)) for row major layout.

The leading dimension of \(b\); at least max(1, \(n\)) for column major layout and at least max(1, \(p\)) for row major layout.

**Output Parameters**

Overwritten by the factorization data as follows:

on exit, the elements on and above the diagonal of the array \(a\) contain the \(\min(n,m)\)-by-\(m\) upper trapezoidal matrix \(R\) \((R\) is upper triangular if \(n\geq m\)); the elements below the diagonal, with the array \(\text{taua}\), represent the orthogonal/unitary matrix \(Q\) as a product of \(\min(n,m)\) elementary reflectors;

if \(n\leq p\), the upper triangle of the subarray \(b(1:n, p-n+1:p)\) contains the \(n\)-by-\(n\) upper triangular matrix \(T\);

if \(n > p\), the elements on and above the \((n-p)\)th subdiagonal contain the \(n\)-by-\(p\) upper trapezoidal matrix \(T\); the remaining elements, with the array \(\text{taub}\), represent the orthogonal/unitary matrix \(Z\) as a product of elementary reflectors.

Arrays, size at least max (1, \(\min(n, m)\)) for \(\text{taua}\) and at least max (1, \(\min(n, p)\)) for \(\text{taub}\). The array \(\text{taua}\) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \(Q\).

The array \(\text{taub}\) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \(Z\).

**Return Values**

This function returns a value  \(\text{info}\).

If  \(\text{info}=0\), the execution is successful.

If  \(\text{info} = -i\), the \(i\)-th parameter had an illegal value.

**Application Notes**

The matrix \(Q\) is represented as a product of elementary reflectors

\[ Q = H(1)H(2) \ldots H(k) \text{, where } k = \min(n,m). \]

Each \(H(i)\) has the form

\[ H(i) = I - \tau_a^*v^*v \]  for real flavors, or

\[ H(i) = I - \tau_a^*v^Hv \]  for complex flavors,

where \(\tau_a\) is a real/complex scalar, and \(v\) is a real/complex vector with \(v_j = 0\) for \(1 \leq j \leq i-1\), \(v_i = 1\).

On exit, for \(1 \leq j \leq n\), \(v_j\) is stored in \(a[(j - 1) + (i - 1)*\text{lda}]\) for column major layout and in \(a[(j - 1)*\text{lda} + (i - 1)]\) for row major layout and \(\tau_a\) is stored in \(\text{taua}[i-1]\)

The matrix \(Z\) is represented as a product of elementary reflectors

\[ Z = H(1)H(2) \ldots H(k) \text{, where } k = \min(n,p). \]

Each \(H(i)\) has the form

\[ H(i) = I - \tau_b^*v^*v \]  for real flavors, or

\[ H(i) = I - \tau_b^*v^Hv \]  for complex flavors,
where $\tau_b$ is a real/complex scalar, and $v$ is a real/complex vector with $v_{p-k+1} = 1$, $v_j = 0$ for $p-k+1 < j < p$.

On exit, for $1 \leq j \leq p-k+i-1$, $v_j$ is stored in $b[(n-k+i-1) + (j-1)*ldb]$ for column major layout and in $b[(n-k+i-1)*ldb + (j-1)]$ for row major layout and $\tau_b$ is stored in $taub[i-1]$.

?qgrqf

Computes the generalized RQ factorization of two matrices.

**Syntax**

```c

lapack_int LAPACKE_sggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n, float* a, lapack_int lda, float* taua, float* b, lapack_int ldb, float* taub);

lapack_int LAPACKE_dggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n, double* a, lapack_int lda, double* taua, double* b, lapack_int ldb, double* taub);

lapack_int LAPACKE_cggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* taua, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* taub);

lapack_int LAPACKE_zggrqf (int matrix_layout, lapack_int m, lapack_int p, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* taua, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* taub);
```

**Include Files**

- mkl.h

**Description**

The routine forms the generalized RQ factorization of an $m$-by-$n$ matrix $A$ and an $p$-by-$n$ matrix $B$ as $A = R \cdot Q$, $B = Z^T \cdot Q$, where $Q$ is an $n$-by-$n$ orthogonal/unitary matrix, $Z$ is a $p$-by-$p$ orthogonal/unitary matrix, and $R$ and $T$ assume one of the forms:

\[
R = \begin{cases} 
  \begin{pmatrix} n-m & m \\ m & R_{12} \end{pmatrix}, & \text{if } m \leq n, \\
  \begin{pmatrix} n \\ m-n \end{pmatrix} \begin{pmatrix} R_{11} \\ R_{21} \end{pmatrix}, & \text{if } m > n, 
\end{cases}
\]

where $R_{11}$ or $R_{21}$ is upper triangular, and

\[
T = \begin{cases} 
  \begin{pmatrix} n \\ p-n \end{pmatrix} T_{11}, & \text{if } p \geq n, \\
  \begin{pmatrix} T_{11} \\ 0 \end{pmatrix}, & \text{else} 
\end{cases}
\]
where $T_{11}$ is upper triangular.

In particular, if $B$ is square and nonsingular, the GRQ factorization of $A$ and $B$ implicitly gives the RQ factorization of $A^*B^{-1}$ as:

\[
A^*B^{-1} = (R^*T^{-1})^*Z^T \text{ (for real flavors) or } A^*B^{-1} = (R^*T^{-1})^*Z^H \text{ (for complex flavors)}.
\]

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **m**
  The number of rows of the matrix $A$ ($m \geq 0$).

- **p**
  The number of rows in $B$ ($p \geq 0$).

- **n**
  The number of columns of the matrices $A$ and $B$ ($n \geq 0$).

- **a, b**
  Arrays:
  - $a$ (size max(1, $lda \times n$) for column major layout and max(1, $lda \times m$) for row major layout) contains the $m$-by-$n$ matrix $A$.
  - $b$ (size max(1, $ldb \times n$) for column major layout and max(1, $ldb \times p$) for row major layout) contains the $p$-by-$n$ matrix $B$.

- **lda**
  The leading dimension of $a$; at least max(1, $m$) for column major layout and max(1, $n$) for row major layout.

- **ldb**
  The leading dimension of $b$; at least max(1, $p$) for column major layout and max(1, $n$) for row major layout.

**Output Parameters**

- **a, b**
  Overwritten by the factorization data as follows:
  - on exit, if $m \leq n$, element $R_{i,j} (1 \leq i \leq n, 1 \leq j \leq m)$ of upper triangular matrix $R$ is stored in $a[(i-1) + (n-m+j-1) \times lda]$ for column major layout and in $a[(i-1) \times lda + (n-m+j-1)]$ for row major layout.
  - if $m > n$, the elements on and above the $(m-n)$th subdiagonal contain the $m$-by-$n$ upper trapezoidal matrix $R$;
  - the remaining elements, with the array $taua$, represent the orthogonal/unitary matrix $Q$ as a product of elementary reflectors.

  The elements on and above the diagonal of the array $b$ contain the min($p,n$)-by-$n$ upper trapezoidal matrix $T$ ($T$ is upper triangular if $p \geq n$); the elements below the diagonal, with the array $taub$, represent the orthogonal/unitary matrix $Z$ as a product of elementary reflectors.

- **taua, taub**
  Arrays, size at least max(1, min($m, n$)) for $taua$ and at least max(1, min($p, n$)) for $taub$.

  The array $taua$ contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix $Q$. 
The array \( taub \) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Z \).

Return Values
This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

Application Notes
The matrix \( Q \) is represented as a product of elementary reflectors
\[
Q = H(1)H(2)\ldots H(k), \text{ where } k = \min(m,n).
\]
Each \( H(i) \) has the form
\[
H(i) = I - taua v^* v^T \text{ for real flavors, or}
H(i) = I - taub v^* v^H \text{ for complex flavors},
\]
where \( taua \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v_{n-k+i} = 1, v_{n-k+i+1:n} = 0 \).
On exit, \( v_1:n-k+i-1 \) is stored in \( a(m-k+i, 1:n-k+i-1) \) and \( taua \) is stored in \( taua[i-1] \).

The matrix \( Z \) is represented as a product of elementary reflectors
\[
Z = H(1)H(2)\ldots H(k), \text{ where } k = \min(p,n).
\]
Each \( H(i) \) has the form
\[
H(i) = I - taub v^* v^T \text{ for real flavors, or}
H(i) = I - taub v^* v^H \text{ for complex flavors},
\]
where \( taub \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v_{1:i-1} = 0, v_{i} = 1 \).
On exit, \( v_{i+1:p} \) is stored in \( b(i+1:p, i) \) and \( taub \) is stored in \( taub[i-1] \).

\tpqrt

*Computes a blocked QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for \( Q \).*

Syntax

```c
lapack_int LAPACKE_stpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_int nb, float* a, lapack_int lda, float* b, lapack_int ldb, float* t, lapack_int ldt);

lapack_int LAPACKE_dtpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_int nb, double* a, lapack_int lda, double* b, lapack_int ldb, double* t, lapack_int ldt);

lapack_int LAPACKE_ctpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_int nb, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* t, lapack_int ldt);

lapack_int LAPACKE_ztpqrt (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_int nb, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* t, lapack_int ldt);
```
Include Files
• mkl.h

Description
The input matrix $C$ is an $(n+m)$-by-$n$ matrix

$$C = \begin{bmatrix}
A & \text{upper triangular} \\
B & \text{pentagonal}
\end{bmatrix}$$

where $A$ is an $n$-by-$n$ upper triangular matrix, and $B$ is an $m$-by-$n$ pentagonal matrix consisting of an $(m-l)$-by-$n$ rectangular matrix $B1$ on top of an $l$-by-$n$ upper trapezoidal matrix $B2$:

$$B = \begin{bmatrix}
B1 & (m-l)\times n \text{ rectangular} \\
B2 & l\times n \text{ upper trapezoidal}
\end{bmatrix}$$

The upper trapezoidal matrix $B2$ consists of the first $l$ rows of an $n$-by-$n$ upper triangular matrix, where $0 \leq l \leq \min(m,n)$. If $l=0$, $B$ is an $m$-by-$n$ rectangular matrix. If $m=l=n$, $B$ is upper triangular. The elementary reflectors $H(i)$ are stored in the $i$th column below the diagonal in the $(n+m)$-by-$n$ input matrix $C$. The structure of vectors defining the elementary reflectors is illustrated by:

$$\begin{bmatrix}
I & \text{identity} \\
V & \text{pentagonal}
\end{bmatrix}$$

The elements of the unit matrix $I$ are not stored. Thus, $V$ contains all of the necessary information, and is returned in array $b$.

**NOTE**
Note that $V$ has the same form as $B$:

$$V = \begin{bmatrix}
V1 & (m-l)\times n \text{ rectangular} \\
V2 & l\times n \text{ upper trapezoidal}
\end{bmatrix}$$

The columns of $V$ represent the vectors which define the $H(i)$.s.

The number of blocks is $k = \text{ceiling}(n/nb)$, where each block is of order $nb$ except for the last block, which is of order $ib = n - (k-1)*nb$. For each of the $k$ blocks, an upper triangular block reflector factor is computed: $T1, T2, \ldots, Tk$. The $nb$-by-$nb$ ($ib$-by-$ib$ for the last block) $T$s are stored in the $nb$-by-$n$ array $t$ as $t = [T1 T2 \ldots Tk]$.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **m**
  The total number of rows in the matrix $B$ ($m \geq 0$).

- **n**
  The number of columns in $B$ and the order of the triangular matrix $A$ ($n \geq 0$).
The number of rows of the upper trapezoidal part of $B$ ($\min(m, n) \geq l \geq 0$).

The block size to use in the blocked QR factorization ($n \geq nb \geq 1$).

Arrays: $a$ size $lda*n$ contains the $n$-by-$n$ upper triangular matrix $A$.

$b$ size $\max(1, ldb*n)$ for column major layout and $\max(1, ldb*m)$ for row major layout, the pentagonal $m$-by-$n$ matrix $B$. The first $(m-l)$ rows contain the rectangular $B1$ matrix, and the next $l$ rows contain the upper trapezoidal $B2$ matrix.

The leading dimension of $a$; at least $\max(1, n)$.

The leading dimension of $b$; at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

The leading dimension of $t$; at least $nb$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

The elements on and above the diagonal of the array contain the upper triangular matrix $R$.

The pentagonal matrix $V$.

Array, size $ldt*n$ for column major layout and $ldt*nb$ for row major layout.

The upper triangular block reflectors stored in compact form as a sequence of upper triangular blocks.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

?tpmqrt

Applies a real or complex orthogonal matrix obtained from a "triangular-pentagonal" complex block reflector to a general real or complex matrix, which consists of two blocks.

Syntax

```c
lapack_int LAPACKE_stpmqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const float* v, lapack_int ldv, const float* t, lapack_int ldt, float* a, lapack_int lda, float* b, lapack_int ldb);
```

```c
lapack_int LAPACKE_dtpmqrt (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const double* v, lapack_int ldv, const double* t, lapack_int ldt, double* a, lapack_int lda, double* b, lapack_int ldb);
```
lapack_int LAPACKE_ctpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const lapack_complex_float* v,
lapack_int ldv, const lapack_complex_float* t, lapack_int ldt, lapack_complex_float* a,
lapack_int lda, lapack_complex_float* b, lapack_int ldb);

lapack_int LAPACKE_ztpmqrt (int matrix_layout, char side, char trans, lapack_int m,
lapack_int n, lapack_int k, lapack_int l, lapack_int nb, const lapack_complex_double* v,
lapack_int ldv, const lapack_complex_double* t, lapack_int ldt,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb);

Include Files
• mkl.h

Description

The columns of the pentagonal matrix $V$ contain the elementary reflectors $H(1), H(2), \ldots, H(k)$; $V$ is composed of a rectangular block $V_1$ and a trapezoidal block $V_2$:

$$V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$$

The size of the trapezoidal block $V_2$ is determined by the parameter $l$, where $0 \leq l \leq k$. $V_2$ is upper trapezoidal, consisting of the first $l$ rows of a $k$-by-$k$ upper triangular matrix.

If $l=k$, $V_2$ is upper triangular;

If $l=0$, there is no trapezoidal block, so $V = V_1$ is rectangular.

If $side = 'L'$:

$$C = \begin{bmatrix} A \\ B \end{bmatrix}$$

where $A$ is $k$-by-$n$, $B$ is $m$-by-$n$ and $V$ is $m$-by-$k$.

If $side = 'R'$:

$$C = \begin{bmatrix} A & B \end{bmatrix}$$

where $A$ is $m$-by-$k$, $B$ is $m$-by-$n$ and $V$ is $n$-by-$k$.

The real/complex orthogonal matrix $Q$ is formed from $V$ and $T$.

If $\text{trans}='N'$ and $\text{side}='L'$, $C$ contains $Q \ast C$ on exit.

If $\text{trans}='T'$ and $\text{side}='L'$, $C$ contains $Q^T \ast C$ on exit.

If $\text{trans}='C'$ and $\text{side}='L'$, $C$ contains $Q^H \ast C$ on exit.

If $\text{trans}='N'$ and $\text{side}='R'$, $C$ contains $C \ast Q$ on exit.

If $\text{trans}='T'$ and $\text{side}='R'$, $C$ contains $C \ast Q^T$ on exit.

If $\text{trans}='C'$ and $\text{side}='R'$, $C$ contains $C \ast Q^H$ on exit.
## Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matrix_layout</code></td>
<td>Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).</td>
</tr>
</tbody>
</table>
| `side` | `'L'`: apply $Q$, $Q^T$, or $Q^H$ from the left.  
`'R'`: apply $Q$, $Q^T$, or $Q^H$ from the right. |
| `trans` | `'N'`, no transpose, apply $Q$.  
`'T'`, transpose, apply $Q^T$.  
`'C'`, transpose, apply $Q^H$. |
| `m` | The number of rows in the matrix $B$, ($m \geq 0$). |
| `n` | The number of columns in the matrix $B$, ($n \geq 0$). |
| `k` | The number of elementary reflectors whose product defines the matrix $Q$, ($k \geq 0$). |
| `l` | The order of the trapezoidal part of $V$ ($k \geq l \geq 0$). |
| `nb` | The block size used for the storage of $t$, $k \geq nb \geq 1$. This must be the same value of $nb$ used to generate $t$ in `tpqrt`. |
| `v` | Size $ldv*k$ for column major layout; $ldv*m$ for row major layout and `side` = `'L'`, $ldv*n$ for row major layout and `side` = `'R'`.  
The $i$th column must contain the vector which defines the elementary reflector $H(i)$, for $i = 1,2,\ldots,k$, as returned by `tpqrt` in array argument $b$. |
| `ldv` | The leading dimension of the array $v$.  
If `side` = `'L'`, $ldv$ must be at least max(1, $m$) for column major layout and max(1, $k$) for row major layout;  
If `side` = `'R'`, $ldv$ must be at least max(1, $n$) for column major layout and max(1, $k$) for row major layout. |
| `t` | Array, size $ldt*k$ for column major layout and $ldt*nb$ for row major layout.  
The upper triangular factors of the block reflectors as returned by `tpqrt`. |
| `ldt` | The leading dimension of the array $t$. $ldt$ must be at least $nb$ for column major layout and max(1, $k$) for row major layout. |
| `a` | If `side` = `'L'`, size $lda*n$ for column major layout and $lda*k$ for row major layout.  
If `side` = `'R'`, size $lda*k$ for column major layout and $lda*m$ for row major layout.  
The $k$-by-$n$ or $m$-by-$k$ matrix $A$. |
| `lda` | The leading dimension of the array $a$.  
If `side` = `'L'`, $lda$ must be at least max(1, $k$) for column major layout and max(1, $n$) for row major layout. |
If `side = 'R'`, `lda` must be at least \( \max(1, m) \) for column major layout and \( \max(1, k) \) for row major layout.

\( b \)
Size \( ldb \times n \) for column major layout and \( ldb \times m \) for row major layout.

The \( n \)-by-\( n \) matrix \( B \).

\( ldb \)
The leading dimension of the array \( b \). \( ldb \) must be at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

**Output Parameters**

\( a \)
Overwritten by the corresponding block of the product \( Q^*C, C^*Q, Q^T*C, C^*Q^T, Q^H*C, \) or \( C^*Q^H \).

\( b \)
Overwritten by the corresponding block of the product \( Q^*C, C^*Q, Q^T*C, C^*Q^T, Q^H*C, \) or \( C^*Q^H \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

**Singular Value Decomposition: LAPACK Computational Routines**

This section describes LAPACK routines for computing the singular value decomposition (SVD) of a general \( m \)-by-\( n \) matrix \( A \):

\[
A = U \Sigma V^H.
\]

In this decomposition, \( U \) and \( V \) are unitary (for complex \( A \)) or orthogonal (for real \( A \)); \( \Sigma \) is an \( m \)-by-\( n \) diagonal matrix with real diagonal elements \( \sigma_i \):

\[
\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{\min(m,n)} \geq 0.
\]

The diagonal elements \( \sigma_i \) are singular values of \( A \). The first \( \min(m, n) \) columns of the matrices \( U \) and \( V \) are, respectively, left and right singular vectors of \( A \). The singular values and singular vectors satisfy

\[
A u_i = \sigma_i u_i \quad \text{and} \quad A^H v_i = \sigma_i v_i
\]

where \( u_i \) and \( v_i \) are the \( i \)-th columns of \( U \) and \( V \), respectively.

To find the SVD of a general matrix \( A \), call the LAPACK routine \(?gebrd\) or \(?gbbrd\) for reducing \( A \) to a bidiagonal matrix \( B \) by a unitary (orthogonal) transformation: \( A = QBP^H \). Then call \(?bdsqr\), which forms the SVD of a bidiagonal matrix: \( B = U \Sigma V^H \).

Thus, the sought-for SVD of \( A \) is given by \( A = U \Sigma V^H = (QU_i) \Sigma (V_j P_j^H) \).

Table "Computational Routines for Singular Value Decomposition (SVD)" lists LAPACK routines that perform singular value decomposition of matrices.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real matrices</th>
<th>Complex matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce ( A ) to a bidiagonal matrix ( B ): ( A = QBP^H ) (full storage)</td>
<td>(?gebrd)</td>
<td>(?gebrd)</td>
</tr>
<tr>
<td>Reduce ( A ) to a bidiagonal matrix ( B ): ( A = QBP^H ) (band storage)</td>
<td>(?gbbrd)</td>
<td>(?gbbrd)</td>
</tr>
</tbody>
</table>
### Decision Tree: Singular Value Decomposition

The table below summarizes the routines for generating orthogonal (unitary) matrices, applying them to form singular value decompositions, and performing related operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real matrices</th>
<th>Complex matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate the orthogonal (unitary) matrix $Q$ or $P$</td>
<td>?orgbr</td>
<td>?ungbr</td>
</tr>
<tr>
<td>Apply the orthogonal (unitary) matrix $Q$ or $P$</td>
<td>?ormbr</td>
<td>?unmbr</td>
</tr>
<tr>
<td>Form singular value decomposition of the bidiagonal matrix $B$: $B = U\Sigma V^H$</td>
<td>?bdsqr ?bdsdc</td>
<td>?bdsqr</td>
</tr>
</tbody>
</table>

**Syntax**

```c
lapack_int LAPACKE_sgebrd( int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* d, float* e, float* tauq, float* taup );

lapack_int LAPACKE_dgebrd( int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* d, double* e, double* tauq, double* taup );
```

**Figure "Decision Tree: Singular Value Decomposition"** presents a decision tree that helps you choose the right sequence of routines for SVD, depending on whether you need singular values only or singular vectors as well, whether $A$ is real or complex, and so on.

You can use the SVD to find a minimum-norm solution to a (possibly) rank-deficient least squares problem of minimizing $||Ax - b||^2$. The effective rank $k$ of the matrix $A$ can be determined as the number of singular values which exceed a suitable threshold. The minimum-norm solution is

$$x = V_k (\Sigma_k)^{-1} c$$

where $\Sigma_k$ is the leading $k$-by-$k$ submatrix of $\Sigma$, the matrix $V_k$ consists of the first $k$ columns of $V = PV_1$, and the vector $c$ consists of the first $k$ elements of $U^H b = U_1^H C_1 b$.

**?gebrd**

*Reduces a general matrix to bidiagonal form.*
lapack_int LAPACKE_cgebrd( int matrix_layout, lapack_int m, lapack_int n, 
apack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float* 
tauq, lapack_complex_float* taup );

lapack_int LAPACKE_zgebrd( int matrix_layout, lapack_int m, lapack_int n, 
apack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double* 
tauq, lapack_complex_double* taup );

Include Files

• mkl.h

Description

The routine reduces a general m-by-n matrix A to a bidiagonal matrix B by an orthogonal (unitary) transformation.

If m≥n, the reduction is given by \( A = QBP^H = \begin{pmatrix} B_1 \end{pmatrix}^H \),

where \( B_1 \) is an n-by-n upper diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; 
\( Q_1 \) consists of the first \( n \) columns of Q.

If \( m < n \), the reduction is given by

\[ A = Q*B*P^H = Q_1*(B_1*0)*P^H = Q_1*B_1*P_1^H, \]

where \( B_1 \) is an m-by-m lower diagonal matrix, Q and P are orthogonal or, for a complex A, unitary matrices; 
\( P_1 \) consists of the first \( m \) columns of P.

The routine does not form the matrices Q and P explicitly, but represents them as products of elementary reflectors. Routines are provided to work with the matrices Q and P in this representation:

If the matrix A is real,

• to compute Q and P explicitly, call orgbr.
• to multiply a general matrix by Q or P, call ormbr.

If the matrix A is complex,

• to compute Q and P explicitly, call ungbr.
• to multiply a general matrix by Q or P, call unmbrr.

Input Parameters

\( \text{matrix\_layout} \quad \text{Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).} \)

\( m \quad \text{The number of rows in the matrix A (} m \geq 0). \)

\( n \quad \text{The number of columns in A (} n \geq 0). \)

\( a \quad \text{Arrays:} \)

\( a(\text{size max}(1, \text{lda}*n) \text{ for column major layout and max}(1, \text{lda}*m) \text{ for row major layout}) \text{ contains the matrix A.} \)

\( lda \quad \text{The leading dimension of a; at least max}(1, m) \text{ for column major layout and at least max}(1, n) \text{ for row major layout.} \)
Output Parameters

If \( m \geq n \), the diagonal and first super-diagonal of \( a \) are overwritten by the upper bidiagonal matrix \( B \). The elements below the diagonal, with the array \( \text{taug} \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors, and the elements above the first superdiagonal, with the array \( \text{taup} \), represent the orthogonal matrix \( P \) as a product of elementary reflectors.

If \( m < n \), the diagonal and first sub-diagonal of \( a \) are overwritten by the lower bidiagonal matrix \( B \). The elements below the first subdiagonal, with the array \( \text{taug} \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors, and the elements above the diagonal, with the array \( \text{taup} \), represent the orthogonal matrix \( P \) as a product of elementary reflectors.

\( d \)

Array, size at least \( \max(1, \min(m, n)) \).
Contains the diagonal elements of \( B \).

\( e \)

Array, size at least \( \max(1, \min(m, n) - 1) \). Contains the off-diagonal elements of \( B \).

\( \text{taug}, \text{taup} \)

Arrays, size at least \( \max(1, \min(m, n)) \). The scalar factors of the elementary reflectors which represent the orthogonal or unitary matrices \( P \) and \( Q \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.
If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The computed matrices \( Q, B, \) and \( P \) satisfy \( QBP^H = A + E \), where \( \| E \|_2 = c(n)\varepsilon \| A \|_2 \), \( c(n) \) is a modestly increasing function of \( n \), and \( \varepsilon \) is the machine precision.

The approximate number of floating-point operations for real flavors is
\[
\begin{align*}
(4/3)n^2*(3m - n) & \text{ for } m \geq n, \\
(4/3)m^2*(3n - m) & \text{ for } m < n.
\end{align*}
\]

The number of operations for complex flavors is four times greater.

If \( n \) is much less than \( m \), it can be more efficient to first form the \( QR \) factorization of \( A \) by calling \( \text{geqrf} \) and then reduce the factor \( R \) to bidiagonal form. This requires approximately \( 2*n^2*(m + n) \) floating-point operations.

If \( m \) is much less than \( n \), it can be more efficient to first form the \( LQ \) factorization of \( A \) by calling \( \text{gelqf} \) and then reduce the factor \( L \) to bidiagonal form. This requires approximately \( 2*m^2*(m + n) \) floating-point operations.

\( \text{?gbbrd} \)

Reduces a general band matrix to bidiagonal form.
**Syntax**

```c
lapack_int LAPACKE_sgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int ncc, lapack_int kl, lapack_int ku, float* ab, lapack_int ldab, float* d, float* e, float* q, lapack_int ldq, float* pt, lapack_int ldpt, float* c, lapack_int ldc);

lapack_int LAPACKE_dgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int ncc, lapack_int kl, lapack_int ku, double* ab, lapack_int ldab, double* d, double* e, double* q, lapack_int ldq, double* pt, lapack_int ldpt, double* c, lapack_int ldc);

lapack_int LAPACKE_cgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_float* ab, lapack_int ldab, float* d, float* e, lapack_complex_float* q, lapack_int ldq, lapack_complex_float* pt, lapack_int ldpt, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zgbbrd( int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int ncc, lapack_int kl, lapack_int ku, lapack_complex_double* ab, lapack_int ldab, double* d, double* e, lapack_complex_double* q, lapack_int ldq, lapack_complex_double* pt, lapack_int ldpt, lapack_complex_double* c, lapack_int ldc);
```

**Include Files**

- `mkl.h`

**Description**

The routine reduces an \( m \times n \) band matrix \( A \) to upper bidiagonal matrix \( B \): \( A = Q^*B^*P^H \). Here the matrices \( Q \) and \( P \) are orthogonal (for real \( A \)) or unitary (for complex \( A \)). They are determined as products of Givens rotation matrices, and may be formed explicitly by the routine if required. The routine can also update a matrix \( C \) as follows: \( C = Q^*C \).

**Input Parameters**

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `vect`
  Must be 'N' or 'Q' or 'P' or 'B'.
  - If `vect = 'N'`, neither \( Q \) nor \( P^H \) is generated.
  - If `vect = 'Q'`, the routine generates the matrix \( Q \).
  - If `vect = 'P'`, the routine generates the matrix \( P^H \).
  - If `vect = 'B'`, the routine generates both \( Q \) and \( P^H \).

- `m`
  The number of rows in the matrix \( A \) \((m \geq 0)\).

- `n`
  The number of columns in \( A \) \((n \geq 0)\).

- `ncc`
  The number of columns in \( C \) \((ncc \geq 0)\).

- `kl`
  The number of sub-diagonals within the band of \( A \) \((kl \geq 0)\).

- `ku`
  The number of super-diagonals within the band of \( A \) \((ku \geq 0)\).

**Arrays**
$ab$ (size max(1, $ldab*n$) for column major layout and max(1, $ldab*m$) for row major layout) contains the matrix $A$ in band storage (see Matrix Storage Schemes).

c (size max(1, $ldc*ncc$) for column major layout and max(1, $ldc*m$) for row major layout) contains an $m$-by-$ncc$ matrix $C$.

If $ncc = 0$, the array $c$ is not referenced.

$ldab$  
The leading dimension of the array $ab$ ($ldab \geq kl + ku + 1$).

$ldq$  
The leading dimension of the output array $q$.

$ldpt$  
The leading dimension of the output array $pt$.

$ldc$  
The leading dimension of the array $c$.

$ldc \geq \max(1, m)$ if $ncc > 0$; $ldc \geq 1$ if $ncc = 0$.

### Output Parameters

$ab$  
Overwritten by values generated during the reduction.

$d$  
Array, size at least max(1, $\min(m, n)$). Contains the diagonal elements of the matrix $B$.

$e$  
Array, size at least $\max(1, \min(m, n) - 1)$. Contains the off-diagonal elements of $B$.

$q$, $pt$  
Arrays:

$q$ size max(1, $ldq*m$) contains the output $m$-by-$m$ matrix $Q$.

$pt$ size max(1, $ldpt*n$) contains the output $n$-by-$n$ matrix $P^T$.

$c$  
Overwritten by the product $Q^H*C$.

$c$ is not referenced if $ncc = 0$.

### Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

### Application Notes

The computed matrices $Q$, $B$, and $P$ satisfy $Q*B*P^H = A + E$, where $\|E\|_2 = c(n)\varepsilon \|A\|_2$, $c(n)$ is a modestly increasing function of $n$, and $\varepsilon$ is the machine precision.

If $m = n$, the total number of floating-point operations for real flavors is approximately the sum of:

$6*n^2*(kl + ku)$ if $vect = 'N'$ and $ncc = 0$,

$3*n^2*ncc*(kl + ku - 1)/(kl + ku)$ if $C$ is updated, and

$3*n^3*(kl + ku - 1)/(kl + ku)$ if either $Q$ or $P^H$ is generated (double this if both).
To estimate the number of operations for complex flavors, use the same formulas with the coefficients 20 and 10 (instead of 6 and 3).

?orgbr
Generates the real orthogonal matrix Q or \( P^T \) determined by ?gebrd.

**Syntax**

```c
lapack_int LAPACKE_sorgbr (int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int k, float* a, lapack_int lda, const float* tau);
lapack_int LAPACKE_dorgbr (int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int k, double* a, lapack_int lda, const double* tau);
```

**Include Files**
- mkl.h

**Description**

The routine generates the whole or part of the orthogonal matrices \( Q \) and \( P^T \) formed by the routines ?gebrd. Use this routine after a call to ?gebrd. All valid combinations of arguments are described in *Input parameters*. In most cases you need the following:

To compute the whole \( m \)-by-\( m \) matrix \( Q \):

```c
LAPACKE_sorgbr(matrix_layout, 'Q', m, m, n, a, lda, tau )
```

(note that the array \( a \) must have at least \( m \) columns).

To form the \( n \) leading columns of \( Q \) if \( m > n \):

```c
LAPACKE_sorgbr(matrix_layout, 'Q', m, n, n, a, lda, tau )
```

To compute the whole \( n \)-by-\( n \) matrix \( P^T \):

```c
LAPACKE_sorgbr(matrix_layout, 'P', n, n, m, a, lda, tau )
```

(note that the array \( a \) must have at least \( n \) rows).

To form the \( m \) leading rows of \( P^T \) if \( m < n \):

```c
LAPACKE_sorgbr(matrix_layout, 'P', m, n, m, a, lda, tau )
```

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **vect**
  Must be 'Q' or 'P'.
  If \( vect = 'Q' \), the routine generates the matrix \( Q \).
  If \( vect = 'P' \), the routine generates the matrix \( P^T \).

- **m, n**
  The number of rows (\( m \)) and columns (\( n \)) in the matrix \( Q \) or \( P^T \) to be returned (\( m \geq 0, n \geq 0 \)).
  If \( vect = 'Q' \), \( m \geq n \geq \text{min}(m, k) \).
  If \( vect = 'P' \), \( n \geq m \geq \text{min}(n, k) \).
If $\text{vect} = 'Q'$, the number of columns in the original $m$-by-$k$ matrix reduced by \texttt{gebrd}.

If $\text{vect} = 'P'$, the number of rows in the original $k$-by-$n$ matrix reduced by \texttt{gebrd}.

$a$ Array, size at least $lda\times n$ for column major layout and $lda\times m$ for row major layout. The vectors which define the elementary reflectors, as returned by \texttt{gebrd}.

$lda$ The leading dimension of the array $a$. $lda \geq \max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

$\tau$ Array, size $\min(m, k)$ if $\text{vect} = 'Q'$, $\min(n, k)$ if $\text{vect} = 'P'$.
Scalar factor of the elementary reflector $H(i)$ or $G(i)$, which determines $Q$ and $P^T$ as returned by \texttt{gebrd} in the array $tauq$ or $taup$.

**Output Parameters**

$a$ Overwritten by the orthogonal matrix $Q$ or $P^T$ (or the leading rows or columns thereof) as specified by $\text{vect}$, $m$, and $n$.

**Return Values**

This function returns a value $\text{info}$.

If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

The computed matrix $Q$ differs from an exactly orthogonal matrix by a matrix $E$ such that $\|E\|_2 = O(\varepsilon)$.

The approximate numbers of floating-point operations for the cases listed in Description are as follows:

To form the whole of $Q$:

\[(4/3)\times n\times (3m^2 - 3m\times n + n^2) \quad \text{if} \quad m > n;\]

\[(4/3)\times m^3 \quad \text{if} \quad m \leq n.\]

To form the $n$ leading columns of $Q$ when $m > n$:

\[(2/3)\times n^2\times (3m - n) \quad \text{if} \quad m > n.\]

To form the whole of $P^T$:

\[(4/3)\times n^3 \quad \text{if} \quad m \geq n;\]

\[(4/3)\times m\times (3n^2 - 3m\times n + m^2) \quad \text{if} \quad m < n.\]

To form the $m$ leading columns of $P^T$ when $m < n$:

\[(2/3)\times n^2\times (3m - n) \quad \text{if} \quad m > n.\]

The complex counterpart of this routine is \texttt{ungbr}.

\texttt{ormbr}

*Multiplies an arbitrary real matrix by the real orthogonal matrix $Q$ or $P^T$ determined by \texttt{gebrd}*
Syntax

```c
lapack_int LAPACKE_sormbr (int matrix_layout, char vect, char side, char trans,
    lapack_int m, lapack_int n, lapack_int k, const float* a, lapack_int lda, const float* 
    tau, float* c, lapack_int ldc);
```

```c
lapack_int LAPACKE_dormbr (int matrix_layout, char vect, char side, char trans,
    lapack_int m, lapack_int n, lapack_int k, const double* a, lapack_int lda, const 
    double* tau, double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

Given an arbitrary real matrix \( C \), this routine forms one of the matrix products \( Q^* C \), \( Q^T C \), \( C^* Q \), \( C^* Q^T \), \( P^* C \), \( C^* P \), \( C^* P^T \), where \( Q \) and \( P \) are orthogonal matrices computed by a call to \texttt{gebrd}. The routine overwrites the product on \( C \).

Input Parameters

In the descriptions below, \( r \) denotes the order of \( Q \) or \( P^T \):

If \( \text{side} = 'L' \), \( r = m \); if \( \text{side} = 'R' \), \( r = n \).

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `vect` Must be 'Q' or 'P'.
  - If `vect` = 'Q', then \( Q \) or \( Q^T \) is applied to \( C \).
  - If `vect` = 'P', then \( P \) or \( P^T \) is applied to \( C \).
- `side` Must be 'L' or 'R'.
  - If `side` = 'L', multipliers are applied to \( C \) from the left.
  - If `side` = 'R', they are applied to \( C \) from the right.
- `trans` Must be 'N' or 'T'.
  - If `trans` = 'N', then \( Q \) or \( P \) is applied to \( C \).
  - If `trans` = 'T', then \( Q^T \) or \( P^T \) is applied to \( C \).
- `m` The number of rows in \( C \).
- `n` The number of columns in \( C \).
- `k` One of the dimensions of \( A \) in \texttt{gebrd}:
  - If `vect` = 'Q', the number of columns in \( A \);
  - If `vect` = 'P', the number of rows in \( A \).
  - Constraints: \( m \geq 0 \), \( n \geq 0 \), \( k \geq 0 \).
- `a`, `c` Arrays:
  - `a` is the array \( a \) as returned by \texttt{gebrd}.
  - The size of \( a \) depends on the value of the `matrix_layout`, `vect`, and `side` parameters:
<table>
<thead>
<tr>
<th>matrix_layout</th>
<th>vect</th>
<th>side</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>column major</td>
<td>'Q'</td>
<td>-</td>
<td>max(1, lda*k)</td>
</tr>
<tr>
<td>column major</td>
<td>'P'</td>
<td>'L'</td>
<td>max(1, lda*m)</td>
</tr>
<tr>
<td>column major</td>
<td>'P'</td>
<td>'R'</td>
<td>max(1, lda*n)</td>
</tr>
<tr>
<td>row major</td>
<td>'Q'</td>
<td>'L'</td>
<td>max(1, lda*m)</td>
</tr>
<tr>
<td>row major</td>
<td>'Q'</td>
<td>'R'</td>
<td>max(1, lda*n)</td>
</tr>
<tr>
<td>row major</td>
<td>'P'</td>
<td>-</td>
<td>max(1, lda*k)</td>
</tr>
</tbody>
</table>

\(c\) (size max(1, ldc*n)) for column major layout and max(1, ldc*m) for row major layout) holds the matrix \(C\).

**lda**

The leading dimension of \(a\). Constraints:

\[\text{lda} \geq \max(1, r)\] for column major layout and at least \(\max(1, k)\) for row major layout if \(\text{vect} = 'Q'\);

\[\text{lda} \geq \max(1, \min(r,k))\] for column major layout and at least \(\max(1, r)\) for row major layout if \(\text{vect} = 'P'\).

**ldc**

The leading dimension of \(c\); \(\text{ldc} \geq \max(1, m)\) for column major layout and \(\text{ldc} \geq \max(1, n)\) for row major layout.

**tau**

Array, size at least \(\max(1, \min(r,k))\).

For \(\text{vect} = 'Q'\), the array \(\tau_{aq}\) as returned by \(?\gebrd\). For \(\text{vect} = 'P'\), the array \(\tau_{ap}\) as returned by \(?\gebrd\).

**Output Parameters**

\(c\)

Overwritten by the product \(Q^*C\), \(Q^T*C\), \(C^*Q\), \(C^*Q^T\), \(P^*C\), \(P^T*C\), \(C^*P\), or \(C^*P^T\), as specified by \(\text{vect}, \text{side}, \text{and trans}\).

**Return Values**

This function returns a value \(\text{info}\).

If \(\text{info}=0\), the execution is successful.

If \(\text{info} = -i\), the \(i\)-th parameter had an illegal value.

**Application Notes**

The computed product differs from the exact product by a matrix \(E\) such that \(\|E\|_2 = O(\varepsilon) * \|C\|_2\).

The total number of floating-point operations is approximately

\[2*n*k(2*m - k)\] if \(\text{side} = 'L'\) and \(m \geq k\);

\[2*m*k(2*n - k)\] if \(\text{side} = 'R'\) and \(n \geq k\);

\[2*m^2*n\] if \(\text{side} = 'L'\) and \(m < k\);

\[2*n^2*m\] if \(\text{side} = 'R'\) and \(n < k\).

The complex counterpart of this routine is \(\text{unmbr}\).
?ungbr
Generates the complex unitary matrix Q or PH determined by ?gebrd.

Syntax

```c
lapack_int LAPACKE_cungbr (int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int k, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zungbr (int matrix_layout, char vect, lapack_int m, lapack_int n, lapack_int k, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);
```

Include Files

- mkl.h

Description

The routine generates the whole or part of the unitary matrices Q and PH formed by the routines gebrd. Use this routine after a call to cgebrd/zgebrd. All valid combinations of arguments are described in Input Parameters; in most cases you need the following:

To compute the whole m-by-m matrix Q, use:

```c
LAPACKE_?ungbr(matrix_layout, 'Q', m, m, n, a, lda, tau)
```
(note that the array a must have at least m columns).

To form the n leading columns of Q if m > n, use:

```c
LAPACKE_?ungbr(matrix_layout, 'Q', m, n, n, a, lda, tau)
```

To compute the whole n-by-n matrix PH, use:

```c
LAPACKE_?ungbr(matrix_layout, 'P', n, n, m, a, lda, tau)
```
(note that the array a must have at least n rows).

To form the m leading rows of PH if m < n, use:

```c
LAPACKE_?ungbr(matrix_layout, 'P', m, m, n, a, lda, tau)
```

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **vect**
  Must be 'Q' or 'P'.
  If vect = 'Q', the routine generates the matrix Q.
  If vect = 'P', the routine generates the matrix PH.

- **m**
  The number of required rows of Q or PH.

- **n**
  The number of required columns of Q or PH.

- **k**
  One of the dimensions of A in ?gebrd:
  If vect = 'Q', the number of columns in A;
  If vect = 'P', the number of rows in A.

Constraints: m≥ 0, n≥ 0, k≥ 0.
For \( \text{vect} = 'Q' \): \( k \leq n \leq m \) if \( m > k \), or \( m = n \) if \( m \leq k \).
For \( \text{vect} = 'P' \): \( k \leq m \leq n \) if \( n > k \), or \( m = n \) if \( n \leq k \).

Arrays:

- \( a \)
  - \( a \), size at least \( lda \times n \) for column major layout and \( lda \times m \) for row major layout, is the array \( a \) as returned by \(?\text{gebrd}\).

- \( lda \)
  - The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

- \( \tau \)
  - For \( \text{vect} = 'Q' \), the array \( \tau_q \) as returned by \(?\text{gebrd}\). For \( \text{vect} = 'P' \), the array \( \tau_p \) as returned by \(?\text{gebrd}\).
  - The dimension of \( \tau \) must be at least \( \max(1, \min(m, k)) \) for \( \text{vect} = 'Q' \), or \( \max(1, \min(m, k)) \) for \( \text{vect} = 'P' \).

Output Parameters

- \( a \)
  - Overwritten by the orthogonal matrix \( Q \) or \( P^T \) (or the leading rows or columns thereof) as specified by \( \text{vect} \), \( m \), and \( n \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

The computed matrix \( Q \) differs from an exactly orthogonal matrix by a matrix \( E \) such that \( \|E\|_2 = O(\varepsilon) \).

The approximate numbers of possible floating-point operations are listed below:

To compute the whole matrix \( Q \):

- \((16/3) n(3m^2 - 3m^*n + n^2)\) if \( m > n \);
- \((16/3) m^3 \) if \( m \leq n \).

To form the \( n \) leading columns of \( Q \) when \( m > n \):

- \((8/3) n^2 (3m - n^2)\).

To compute the whole matrix \( P^H \):

- \((16/3) n^3 \) if \( m \geq n \);
- \((16/3) m(3n^2 - 3m^*n + m^2) \) if \( m < n \).

To form the \( m \) leading columns of \( P^H \) when \( m < n \):

- \((8/3) n^2 (3m - n^2) \) if \( m > n \).

The real counterpart of this routine is \text{orgbr}.

?\text{unmbr}

Multiplies an arbitrary complex matrix by the unitary matrix \( Q \) or \( P \) determined by \(?\text{gebrd}\).
Syntax

```c
lapack_int LAPACKE_cunmbr (int matrix_layout, char vect, char side, char trans,
                        lapack_int m, lapack_int n, lapack_int k, const lapack_complex_float* a, lapack_int lda,
                        const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmbr (int matrix_layout, char vect, char side, char trans,
                        lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double* a, lapack_int lda,
                        const lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description

Given an arbitrary complex matrix \( C \), this routine forms one of the matrix products \( Q^*C \), \( Q^H* C \), \( C*Q \), \( C*Q^H \), \( P*C \), \( P^H*C \), \( C*P \), or \( C*P^H \), where \( Q \) and \( P \) are unitary matrices computed by a call to `gebrd/gebrd`. The routine overwrites the product on \( C \).

Input Parameters

In the descriptions below, \( r \) denotes the order of \( Q \) or \( P^H \):

If \( side = 'L' \), \( r = m \); if \( side = 'R' \), \( r = n \).

- **matrix_layout**: Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **vect**: Must be `'Q'` or `'P'`.
  - If `vect = 'Q'`, then \( Q \) or \( Q^H \) is applied to \( C \).
  - If `vect = 'P'`, then \( P \) or \( P^H \) is applied to \( C \).

- **side**: Must be `'L'` or `'R'`.
  - If `side = 'L'`, multipliers are applied to \( C \) from the left.
  - If `side = 'R'`, they are applied to \( C \) from the right.

- **trans**: Must be `'N'` or `'C'`.
  - If `trans = 'N'`, then \( Q \) or \( P \) is applied to \( C \).
  - If `trans = 'C'`, then \( Q^H \) or \( P^H \) is applied to \( C \).

- **m**: The number of rows in \( C \).
- **n**: The number of columns in \( C \).
- **k**: One of the dimensions of \( A \) in `gebrd`:
  - If `vect = 'Q'`, the number of columns in \( A \);
  - If `vect = 'P'`, the number of rows in \( A \).

Constraints: \( m \geq 0 \), \( n \geq 0 \), \( k \geq 0 \).

- **a**, **c**: Arrays:
  - \( a \) is the array \( a \) as returned by `gebrd`.
  - The size of \( a \) depends on the value of the `matrix_layout`, `vect`, and `side` parameters:
<table>
<thead>
<tr>
<th>matrix_layout</th>
<th>vect</th>
<th>side</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>column major</td>
<td>'Q'</td>
<td>-</td>
<td>(\max(1, \text{lda} \times k))</td>
</tr>
<tr>
<td>column major</td>
<td>'P'</td>
<td>'L'</td>
<td>(\max(1, \text{lda} \times m))</td>
</tr>
<tr>
<td>column major</td>
<td>'P'</td>
<td>'R'</td>
<td>(\max(1, \text{lda} \times n))</td>
</tr>
<tr>
<td>row major</td>
<td>'Q'</td>
<td>'L'</td>
<td>(\max(1, \text{lda} \times m))</td>
</tr>
<tr>
<td>row major</td>
<td>'Q'</td>
<td>'R'</td>
<td>(\max(1, \text{lda} \times n))</td>
</tr>
<tr>
<td>row major</td>
<td>'P'</td>
<td>-</td>
<td>(\max(1, \text{lda} \times k))</td>
</tr>
</tbody>
</table>

\(c(\text{size } \max(1, \text{ldc} \times n))\) for column major layout and \(\max(1, \text{ldc} \times m)\) for row major layout holds the matrix \(C\).

\(\text{lda}\)

The leading dimension of \(a\). Constraints:

- \(\text{lda} \geq \max(1, r)\) for column major layout and at least \(\max(1, k)\) for row major layout if \(\text{vect} = 'Q'\);
- \(\text{lda} \geq \max(1, \min(r, k))\) for column major layout and at least \(\max(1, r)\) for row major layout if \(\text{vect} = 'P'\).

\(\text{ldc}\)

The leading dimension of \(C\); \(\text{ldc} \geq \max(1, m)\).

\(\text{tau}\)

Array, size at least \(\max(1, \min(r, k))\).

For \(\text{vect} = 'Q'\), the array \(\text{tauq}\) as returned by ?gebrd. For \(\text{vect} = 'P'\), the array \(\text{taup}\) as returned by ?gebrd.

**Output Parameters**

\(c\)

Overwritten by the product \(Q \times C\), \(Q^H \times C\), \(C \times Q\), \(C \times Q^H\), \(P \times C\), \(P^H \times C\), \(C \times P\), or \(C \times P^H\), as specified by \(\text{vect}, \text{side}, \) and \(\text{trans}\).

**Return Values**

This function returns a value \(\text{info}\).

If \(\text{info}=0\), the execution is successful.

If \(\text{info} = -i\), the \(i\)-th parameter had an illegal value.

**Application Notes**

The computed product differs from the exact product by a matrix \(E\) such that \(\|E\|_2 = O(\epsilon) \times \|C\|_2\).

The total number of floating-point operations is approximately

- \(8 \times n \times k(2 \times m - k)\) if \(\text{side} = 'L'\) and \(m \geq k\);
- \(8 \times m \times k(2 \times n - k)\) if \(\text{side} = 'R'\) and \(n \geq k\);
- \(8 \times m \times n\) if \(\text{side} = 'L'\) and \(m < k\);
- \(8 \times n \times n\) if \(\text{side} = 'R'\) and \(n < k\).
The real counterpart of this routine is ormb.

?bdsqr
Computes the singular value decomposition of a general matrix that has been reduced to bidiagonal form.

Syntax

lapack_int LAPACKE_sbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt, lapack_int nru, lapack_int ncc, float* d, float* e, float* vt, lapack_int ldvt, float* u, lapack_int ldu, float* c, lapack_int ldc );

lapack_int LAPACKE_dbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt, lapack_int nru, lapack_int ncc, double* d, double* e, double* vt, lapack_int ldvt, double* u, lapack_int ldu, double* c, lapack_int ldc );

lapack_int LAPACKE_cbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt, lapack_int nru, lapack_int ncc, float* d, float* e, lapack_complex_float* vt, lapack_int ldvt, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* c, lapack_int ldc );

lapack_int LAPACKE_zbdsqr( int matrix_layout, char uplo, lapack_int n, lapack_int ncvt, lapack_int nru, lapack_int ncc, double* d, double* e, lapack_complex_double* vt, lapack_int ldvt, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* c, lapack_int ldc );

Include Files

- mkl.h

Description

The routine computes the singular values and, optionally, the right and/or left singular vectors from the Singular Value Decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix B using the implicit zero-shift QR algorithm. The SVD of B has the form $B = Q S P^H$ where $S$ is the diagonal matrix of singular values, $Q$ is an orthogonal matrix of left singular vectors, and $P$ is an orthogonal matrix of right singular vectors. If left singular vectors are requested, this subroutine actually returns $U^*Q$ instead of $Q$, and, if right singular vectors are requested, this subroutine returns $P^H*VT$ instead of $P^H$, for given real/complex input matrices $U$ and $VT$. When $U$ and $VT$ are the orthogonal/unitary matrices that reduce a general matrix $A$ to bidiagonal form: $A = U^*B*VT$, as computed by ?gebrd, then

$A = (U^*Q)*S*(P^H*VT)$

is the SVD of $A$. Optionally, the subroutine may also compute $Q^H*C$ for a given real/complex input matrix $C$.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo
Must be 'U' or 'L'.
If uplo = 'U', $B$ is an upper bidiagonal matrix.
If uplo = 'L', $B$ is a lower bidiagonal matrix.

n
The order of the matrix $B$ ($n \geq 0$).
The number of columns of the matrix $VT$, that is, the number of right singular vectors ($ncvt \geq 0$).

Set $ncvt = 0$ if no right singular vectors are required.

The number of rows in $U$, that is, the number of left singular vectors ($nru \geq 0$).

Set $nru = 0$ if no left singular vectors are required.

The number of columns in the matrix $C$ used for computing the product $Q^H \cdot C$ ($ncc \geq 0$). Set $ncc = 0$ if no matrix $C$ is supplied.

Arrays:

$d$, $e$

$d$ contains the diagonal elements of $B$.

The size of $d$ must be at least $\max(1, n)$.

$e$ contains the $(n-1)$ off-diagonal elements of $B$.

The size of $e$ must be at least $\max(1, n - 1)$.

Arrays:

$vt$, $u$, $c$

$vt$, size $\max(1, ldvt*ncvt)$ for column major layout and $\max(1, ldvt*n)$ for row major layout, contains an $n$-by-$ncvt$ matrix $VT$.

$vt$ is not referenced if $ncvt = 0$.

$u$, size $\max(1, ldu*n)$ for column major layout and $\max(1, ldu*nru)$ for row major layout, contains an $nru$ by $n$ matrix $U$.

$u$ is not referenced if $nru = 0$.

$c$, size $\max(1, ldc*ncc)$ for column major layout and $\max(1, ldc*n)$ for row major layout, contains the $n$-by-$ncc$ matrix $C$ for computing the product $Q^H \cdot C$.

The leading dimension of $vt$. Constraints:

$ldvt \geq \max(1, n)$ if $ncvt > 0$ for column major layout and $ldvt \geq \max(1, ncvt)$ for row major layout;

$ldvt \geq 1$ if $ncvt = 0$.

The leading dimension of $u$. Constraint:

$ldu \geq \max(1, nru)$ for column major layout and $ldu \geq \max(1, n)$ for row major layout.

The leading dimension of $c$. Constraints:

$ldc \geq \max(1, n)$ if $ncc > 0$ for column major layout and $ldc \geq \max(1, ncc)$ for row major layout; $ldc \geq 1$ otherwise.

**Output Parameters**

$d$

On exit, if $info = 0$, overwritten by the singular values in decreasing order (see $info$).

$e$

On exit, if $info = 0$, $e$ is destroyed. See also $info$ below.
Overwritten by the product $Q^H*C$.

On exit, this array is overwritten by $P^*VT$. Not referenced if $ncvt = 0$.

On exit, this array is overwritten by $U*Q$. Not referenced if $nru = 0$.

**Return Values**

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info > 0$,

If $ncvt = nru = ncc = 0$,

- $info = 1$, a split was marked by a positive value in $e$
- $info = 2$, the current block of $z$ not diagonalized after $100*n$ iterations (in the inner while loop)
- $info = 3$, termination criterion of the outer while loop is not met (the program created more than $n$ unreduced blocks).

In all other cases when $ncvt$, $nru$, or $ncc > 0$, the algorithm did not converge; $d$ and $e$ contain the elements of a bidiagonal matrix that is orthogonally similar to the input matrix $B$; if $info = i$, $i$ elements of $e$ have not converged to zero.

**Application Notes**

Each singular value and singular vector is computed to high relative accuracy. However, the reduction to bidiagonal form (prior to calling the routine) may decrease the relative accuracy in the small singular values of the original matrix if its singular values vary widely in magnitude.

If $s_i$ is an exact singular value of $B$, and $s_j$ is the corresponding computed value, then

$$|s_i - s_j| \leq p^*(m, n) * \varepsilon * s_i$$

where $p(m, n)$ is a modestly increasing function of $m$ and $n$, and $\varepsilon$ is the machine precision.

If only singular values are computed, they are computed more accurately than when some singular vectors are also computed (that is, the function $p(m, n)$ is smaller).

If $u_i$ is the corresponding exact left singular vector of $B$, and $w_i$ is the corresponding computed left singular vector, then the angle $\theta(u_i, w_i)$ between them is bounded as follows:

$$\theta(u_i, w_i) \leq p(m, n) * \varepsilon / \min_{i \neq j}(|s_i - s_j|/|s_i + s_j|).$$

Here $\min_{i \neq j}(|s_i - s_j|/|s_i + s_j|)$ is the relative gap between $s_i$ and the other singular values. A similar error bound holds for the right singular vectors.

The total number of real floating-point operations is roughly proportional to $n^2$ if only the singular values are computed. About $6n^2*nru$ additional operations ($12n^2*nru$ for complex flavors) are required to compute the left singular vectors and about $6n^2*ncvt$ operations ($12n^2*ncvt$ for complex flavors) to compute the right singular vectors.

?bdsdc

*Computes the singular value decomposition of a real bidiagonal matrix using a divide and conquer method.*
Syntax

```c
lapack_int LAPACKE_sbdsc (int matrix_layout, char uplo, char compq, lapack_int n, float* d, float* e, float* u, lapack_int ldu, float* vt, lapack_int ldvt, float* q, lapack_int* iq);
```

```c
lapack_int LAPACKE_dbdsc (int matrix_layout, char uplo, char compq, lapack_int n, double* d, double* e, double* u, lapack_int ldu, double* vt, lapack_int ldvt, double* q, lapack_int* iq);
```

Include Files
- mkl.h

Description

The routine computes the Singular Value Decomposition (SVD) of a real \( n \)-by-\( n \) (upper or lower) bidiagonal matrix \( B \): \( B = U \Sigma V^T \), using a divide and conquer method, where \( \Sigma \) is a diagonal matrix with non-negative diagonal elements (the singular values of \( B \)), and \( U \) and \( V \) are orthogonal matrices of left and right singular vectors, respectively. \( \text{?bdsdc} \) can be used to compute all singular values, and optionally, singular vectors or singular vectors in compact form.

This routine uses \( \text{?lasd0} \), \( \text{?lasd1} \), \( \text{?lasd2} \), \( \text{?lasd3} \), \( \text{?lasd4} \), \( \text{?lasd5} \), \( \text{?lasd6} \), \( \text{?lasd7} \), \( \text{?lasd8} \), \( \text{?lasd9} \), \( \text{?lasda} \), \( \text{?lasdq} \), \( \text{?lasdt} \).

Input Parameters

- \textit{matrix_layout}
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- \textit{uplo}
  Must be 'U' or 'L'.
  If \( \text{uplo} = 'U' \), \( B \) is an upper bidiagonal matrix.
  If \( \text{uplo} = 'L' \), \( B \) is a lower bidiagonal matrix.

- \textit{compq}
  Must be 'N', 'P', or 'I'.
  If \( \text{compq} = 'N' \), compute singular values only.
  If \( \text{compq} = 'P' \), compute singular values and compute singular vectors in compact form.
  If \( \text{compq} = 'I' \), compute singular values and singular vectors.

- \textit{n}
  The order of the matrix \( B \) (\( n \geq 0 \)).

- \textit{d, e}
  Arrays:
  \( d \) contains the \( n \) diagonal elements of the bidiagonal matrix \( B \). The size of \( d \) must be at least \( \max(1, n) \).
  \( e \) contains the off-diagonal elements of the bidiagonal matrix \( B \). The size of \( e \) must be at least \( \max(1, n) \).

- \textit{ldu}
  The leading dimension of the output array \( u \); \( ldu \geq 1 \).
  If singular vectors are desired, then \( ldu \geq \max(1, n) \), regardless of the value of \textit{matrix_layout}.

- \textit{ldvt}
  The leading dimension of the output array \( vt \); \( ldvt \geq 1 \).
If singular vectors are desired, then $ldvt \geq \max(1, n)$, regardless of the value of $\text{matrix\_layout}$.

**Output Parameters**

- $d$
  - If $info = 0$, overwritten by the singular values of $B$.
  - On exit, $e$ is overwritten.

- $u$, $vt$, $q$
  - Arrays: $u(\text{size } ldu*n)$, $vt(\text{size } ldvt*n)$, $q(\text{size } \geq n*(11 + 2*smlsiz + 8*\text{int}(\log_2(n/(smlsiz+1))))$ where $smlsiz$ is returned by $\text{ilaenv}$ and is equal to maximum size of the subproblems at the bottom of the computation tree.
  - If $\text{compq} = 'I'$, then on exit $u$ contains the left singular vectors of the bidiagonal matrix $B$, unless $info \neq 0$ (see $info$). For other values of $\text{compq}$, $u$ is not referenced.
  - If $\text{compq} = 'I'$, then on exit $vt^T$ contains the right singular vectors of the bidiagonal matrix $B$, unless $info \neq 0$ (see $info$). For other values of $\text{compq}$, $vt$ is not referenced.
  - If $\text{compq} = 'P'$, then on exit, if $info = 0$, $q$ and $iq$ contain the left and right singular vectors in a compact form. Specifically, $q$ contains all the float (for $\text{sbdsdc}$) or double (for $\text{dbdsdc}$) data for singular vectors. For other values of $\text{compq}$, $q$ is not referenced.

- $iq$
  - Array: $iq(\text{size } \geq n*(3 + 3*\text{int}(\log_2(n/(smlsiz+1))))$ where $smlsiz$ is returned by $\text{ilaenv}$ and is equal to maximum size of the subproblems at the bottom of the computation tree.
  - If $\text{compq} = 'P'$, then on exit, if $info = 0$, $q$ and $iq$ contain the left and right singular vectors in a compact form. Specifically, $iq$ contains all the lapack_int data for singular vectors. For other values of $\text{compq}$, $iq$ is not referenced.

**Return Values**

This function returns a value $info$.

- If $info = 0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.
- If $info = i$, the algorithm failed to compute a singular value. The update process of divide and conquer failed.

**Symmetric Eigenvalue Problems: LAPACK Computational Routines**

Symmetric eigenvalue problems are posed as follows: given an $n$-by-$n$ real symmetric or complex Hermitian matrix $A$, find the eigenvalues $\lambda$ and the corresponding eigenvectors $z$ that satisfy the equation $Az = \lambda z$ (or, equivalently, $z^H A = \lambda z^H$).

In such eigenvalue problems, all $n$ eigenvalues are real not only for real symmetric but also for complex Hermitian matrices $A$, and there exists an orthonormal system of $n$ eigenvectors. If $A$ is a symmetric or Hermitian positive-definite matrix, all eigenvalues are positive.
To solve a symmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to tridiagonal form and then solve the eigenvalue problem with the tridiagonal matrix obtained. LAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation $A = QTQ^H$ as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table "Computational Routines for Solving Symmetric Eigenvalue Problems".

There are different routines for symmetric eigenvalue problems, depending on whether you need all eigenvectors or only some of them or eigenvalues only, whether the matrix $A$ is positive-definite or not, and so on.

These routines are based on three primary algorithms for computing eigenvalues and eigenvectors of symmetric problems: the divide and conquer algorithm, the QR algorithm, and bisection followed by inverse iteration. The divide and conquer algorithm is generally more efficient and is recommended for computing all eigenvalues and eigenvectors. Furthermore, to solve an eigenvalue problem using the divide and conquer algorithm, you need to call only one routine. In general, more than one routine has to be called if the QR algorithm or bisection followed by inverse iteration is used.

**Computational Routines for Solving Symmetric Eigenvalue Problems**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real symmetric matrices</th>
<th>Complex Hermitian matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce to tridiagonal form $A = QTQ^H$ (full storage)</td>
<td>sytrd</td>
<td>hetrd</td>
</tr>
<tr>
<td>Reduce to tridiagonal form $A = QTQ^H$ (packed storage)</td>
<td>sptrd</td>
<td>hptrd</td>
</tr>
<tr>
<td>Reduce to tridiagonal form $A = QTQ^H$ (band storage)</td>
<td>sbtrd</td>
<td>hbtrd</td>
</tr>
<tr>
<td>Generate matrix $Q$ (full storage)</td>
<td>orgtr</td>
<td>ungtr</td>
</tr>
<tr>
<td>Generate matrix $Q$ (packed storage)</td>
<td>opgtr</td>
<td>upgtr</td>
</tr>
<tr>
<td>Apply matrix $Q$ (full storage)</td>
<td>ormtr</td>
<td>unmtr</td>
</tr>
<tr>
<td>Apply matrix $Q$ (packed storage)</td>
<td>opmtr</td>
<td>upmtr</td>
</tr>
<tr>
<td>Find all eigenvalues of a tridiagonal matrix $T$</td>
<td>sterf</td>
<td></td>
</tr>
<tr>
<td>Find all eigenvalues and eigenvectors of a tridiagonal matrix $T$</td>
<td>steqr, stedc</td>
<td>steqr, stedc</td>
</tr>
<tr>
<td>Find all eigenvalues and eigenvectors of a tridiagonal positive-definite matrix $T$.</td>
<td>pteqr</td>
<td>pteqr</td>
</tr>
<tr>
<td>Find selected eigenvalues of a tridiagonal matrix $T$</td>
<td>stebz, stegr</td>
<td>stegr</td>
</tr>
<tr>
<td>Find selected eigenvectors of a tridiagonal matrix $T$</td>
<td>stein, stegr</td>
<td>stein, stegr</td>
</tr>
<tr>
<td>Find selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix $T$</td>
<td>stemr</td>
<td>stemr</td>
</tr>
<tr>
<td>Compute the reciprocal condition numbers for the eigenvectors</td>
<td>disna</td>
<td>disna</td>
</tr>
</tbody>
</table>

**?sytrd**

*Reduces a real symmetric matrix to tridiagonal form.*
Syntax

```c
lapack_int LAPACKE_ssytrd (int matrix_layout, char uplo, lapack_int n, float* a,
                        lapack_int lda, float* d, float* e, float* tau);

lapack_int LAPACKE_dsytrd (int matrix_layout, char uplo, lapack_int n, double* a,
                        lapack_int lda, double* d, double* e, double* tau);
```

Include Files

- mkl.h

Description

The routine reduces a real symmetric matrix \( A \) to symmetric tridiagonal form \( T \) by an orthogonal similarity transformation: \( A = Q^TQ^2 \). The orthogonal matrix \( Q \) is not formed explicitly but is represented as a product of \( n-1 \) elementary reflectors. Routines are provided for working with \( Q \) in this representation (see Application Notes below).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be ‘U’ or ‘L’.
  If \( uplo = 'U' \), \( A \) stores the upper triangular part of \( A \).
  If \( uplo = 'L' \), \( A \) stores the lower triangular part of \( A \).

- **n**
  The order of the matrix \( A \) (\( n \geq 0 \)).

- **a**
  \( a \) (size max(1, lda\(n\))) is an array containing either upper or lower triangular part of the matrix \( A \), as specified by \( uplo \). If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( a \) contains the upper triangular part of the matrix \( A \), and the strictly lower triangular part of \( A \) is not referenced. If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( a \) contains the lower triangular part of the matrix \( A \), and the strictly upper triangular part of \( A \) is not referenced.

- **lda**
  The leading dimension of \( a \); at least max(1, \( n \)).

Output Parameters

- **a**
  On exit, if \( uplo = 'U' \), the diagonal and first superdiagonal of \( A \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements above the first superdiagonal, with the array \( tau \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors;
  if \( uplo = 'L' \), the diagonal and first subdiagonal of \( A \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements below the first subdiagonal, with the array \( tau \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors.

- **d**, **e**, **tau**
  Arrays:
  - \( d \) contains the diagonal elements of the matrix \( T \).
The size of \( d \) must be at least \( \max(1, n) \).

\( e \) contains the off-diagonal elements of \( T \).

The size of \( e \) must be at least \( \max(1, n-1) \).

\( \tau \) stores \((n-1)\) scalars that define elementary reflectors in decomposition of the orthogonal matrix \( Q \) in a product of \( n-1 \) elementary reflectors. \( \tau(n) \) is used as workspace.

The size of \( \tau \) must be at least \( \max(1, n) \).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The computed matrix \( T \) is exactly similar to a matrix \( A+E \), where \( ||E||_2 = c(n) \cdot \epsilon \cdot ||A||_2 \), \( c(n) \) is a modestly increasing function of \( n \), and \( \epsilon \) is the machine precision.

The approximate number of floating-point operations is \((4/3)n^3\).

After calling this routine, you can call the following:

- \( \text{orgtr} \) to form the computed matrix \( Q \) explicitly
- \( \text{ormtr} \) to multiply a real matrix by \( Q \).

The complex counterpart of this routine is \( \text{?hetrd} \).

\( \text{?orgtr} \)

Generates the real orthogonal matrix \( Q \) determined by \( \text{?sytrd} \).

**Syntax**

```c
lapack_int LAPACKE_sorgtr (int matrix_layout, char uplo, lapack_int n, float* a, lapack_int lda, const float* tau);

lapack_int LAPACKE_dorgtr (int matrix_layout, char uplo, lapack_int n, double* a, lapack_int lda, const double* tau);
```

**Include Files**

- mkl.h

**Description**

The routine explicitly generates the \( n \)-by-\( n \) orthogonal matrix \( Q \) formed by \( \text{?sytrd} \) when reducing a real symmetric matrix \( A \) to tridiagonal form: \( A = Q^T T Q \). Use this routine after a call to \( \text{?sytrd} \).

**Input Parameters**

- \( \text{matrix_layout} \)
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- \( \text{uplo} \)
  
  Must be ‘U’ or ‘L’.
Use the same \textit{uplo} as supplied to \texttt{?sytrd}.

\texttt{n} \\

The order of the matrix \( Q (n \geq 0) \).

\texttt{a, tau} \\

Arrays:

\texttt{a} (size max(1, \texttt{lda}*n)) is the array \( a \) as returned by \texttt{?sytrd}.

\texttt{tau} is the array \( \tau \) as returned by \texttt{?sytrd}.

The size of \( \tau \) must be at least max(1, \( n-1 \)).

\texttt{lda} \\

The leading dimension of \( a \); at least max(1, \( n \)).

\textbf{Output Parameters} \\

\texttt{a} \\

Overwritten by the orthogonal matrix \( Q \).

\textbf{Return Values} \\

This function returns a value \( \texttt{info} \).

If \( \texttt{info}=0 \), the execution is successful.

If \( \texttt{info} = -i \), the \( i \)-th parameter had an illegal value.

\textbf{Application Notes} \\

The computed matrix \( Q \) differs from an exactly orthogonal matrix by a matrix \( E \) such that \(||E||_2 = O(\epsilon)\), where \( \epsilon \) is the machine precision.

The approximate number of floating-point operations is \((4/3)n^3\).

The complex counterpart of this routine is \texttt{ungtr}.

\texttt{?ormtr} \texttt{Multiplies a real matrix by the real orthogonal matrix} \\
\texttt{Q determined by \texttt{?sytrd}.} \\

\textbf{Syntax} \\

\begin{verbatim}
lapack_int LAPACKE_sormtr (int matrix_layout, char side, char uplo, char trans, 
lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* tau, float* c, 
lapack_int ldc);

lapack_int LAPACKE_dormtr (int matrix_layout, char side, char uplo, char trans, 
lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* tau, double* c, 
lapack_int ldc);
\end{verbatim}

\textbf{Include Files} \\

\begin{itemize}
  \item mkl.h
\end{itemize}

\textbf{Description} \\

The routine multiplies a real matrix \( C \) by \( Q \) or \( Q^T \), where \( Q \) is the orthogonal matrix \( Q \) formed by \texttt{sytrd} when reducing a real symmetric matrix \( A \) to tridiagonal form: \( A = Q^T \Sigma Q \). Use this routine after a call to \texttt{?sytrd}.

Depending on the parameters \texttt{side} and \texttt{trans}, the routine can form one of the matrix products \( Q^T C, Q^T Q, \) \( C \), or \( C^T Q \) (overwriting the result on \( C \)).
Input Parameters

In the descriptions below, $r$ denotes the order of $Q$:

If $side = 'L'$, $r = m$; if $side = 'R'$, $r = n$.

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **side**: Must be either 'L' or 'R'.
  - If $side = 'L'$, $Q$ or $Q^T$ is applied to $C$ from the left.
  - If $side = 'R'$, $Q$ or $Q^T$ is applied to $C$ from the right.

- **uplo**: Must be 'U' or 'L'.
  - Use the same uplo as supplied to ?sytrd.

- **trans**: Must be either 'N' or 'T'.
  - If $trans = 'N'$, the routine multiplies $C$ by $Q$.
  - If $trans = 'T'$, the routine multiplies $C$ by $Q^T$.

- **m**: The number of rows in the matrix $C$ ($m \geq 0$).

- **n**: The number of columns in $C$ ($n \geq 0$).

- **a, c, tau**: $a$ (size $\max(1, lda \times r)$) and $tau$ are the arrays returned by ?sytrd.
  - The size of $tau$ must be at least $\max(1, r-1)$.
  - $c$ (size $\max(1, ldc \times n)$ for column major layout and $\max(1, ldc \times m)$ for row major layout) contains the matrix $C$.

- **lda**: The leading dimension of $a$; $lda \geq \max(1, r)$.

- **ldc**: The leading dimension of $c$; $ldc \geq \max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

Output Parameters

- **c**: Overwritten by the product $Q \times C$, $Q^T \times C$, $C \times Q$, or $C \times Q^T$ (as specified by $side$ and $trans$).

Return Values

This function returns a value $info$.

- If $info=0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

- The computed product differs from the exact product by a matrix $E$ such that $||E||_2 = O(\varepsilon) \times ||C||_2$.

- The total number of floating-point operations is approximately $2 \times m^2 \times n$, if $side = 'L'$, or $2 \times n^2 \times m$, if $side = 'R'$.

- The complex counterpart of this routine is unmtr.
Hetrd
Reduces a complex Hermitian matrix to tridiagonal form.

Syntax

```c
lapack_int LAPACKE_chetrd( int matrix_layout, char uplo, lapack_int n,
                         lapack_complex_float* a, lapack_int lda, float* d, float* e, lapack_complex_float* tau );

lapack_int LAPACKE_zhetrd( int matrix_layout, char uplo, lapack_int n,
                         lapack_complex_double* a, lapack_int lda, double* d, double* e, lapack_complex_double* tau );
```

Include Files

- mkl.h

Description

The routine reduces a complex Hermitian matrix $A$ to symmetric tridiagonal form $T$ by a unitary similarity transformation: $A = Q^*TQ$. The unitary matrix $Q$ is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided to work with $Q$ in this representation.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `uplo` Must be 'U' or 'L'.

  If `uplo` = 'U', $a$ stores the upper triangular part of $A$.

  If `uplo` = 'L', $a$ stores the lower triangular part of $A$.

- `n` The order of the matrix $A$ ($n \geq 0$).

- `a` ($\text{size max}(1, \text{lda} \times n)$) is an array containing either upper or lower triangular part of the matrix $A$, as specified by `uplo`. If `uplo` = 'U', the leading $n$-by-$n$ upper triangular part of $a$ contains the upper triangular part of the matrix $A$, and the strictly lower triangular part of $A$ is not referenced. If `uplo` = 'L', the leading $n$-by-$n$ lower triangular part of $a$ contains the lower triangular part of the matrix $A$, and the strictly upper triangular part of $A$ is not referenced.

  The leading dimension of $a$; at least max(1, $n$).

Output Parameters

- `a`

  On exit, if `uplo` = 'U', the diagonal and first superdiagonal of $A$ are overwritten by the corresponding elements of the tridiagonal matrix $T$, and the elements above the first superdiagonal, with the array `tau`, represent the orthogonal matrix $Q$ as a product of elementary reflectors;
if uplo = 'L', the diagonal and first subdiagonal of A are overwritten by
the corresponding elements of the tridiagonal matrix T, and the elements
below the first subdiagonal, with the array tau, represent the orthogonal
matrix Q as a product of elementary reflectors.

d, e

Arrays:
d contains the diagonal elements of the matrix T.
The dimension of d must be at least max(1, n).
e contains the off-diagonal elements of T.
The dimension of e must be at least max(1, n-1).

tau

Array, size at least max(1, n-1). Stores (n-1) scalars that define elementary
reflectors in decomposition of the unitary matrix Q in a product of n-1
elementary reflectors.

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes
The computed matrix T is exactly similar to a matrix A + E, where ||E||_2 = c(n)*\epsilon*||A||_2, c(n) is a
modestly increasing function of n, and \epsilon is the machine precision.
The approximate number of floating-point operations is (16/3) n^3.
After calling this routine, you can call the following:
    ungtr              to form the computed matrix Q explicitly
    unmtr              to multiply a complex matrix by Q.
The real counterpart of this routine is ?sytrd.

?ungtr
Generates the complex unitary matrix Q determined
by ?hetrd.

Syntax
lapack_int LAPACKE_cungtr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);  
lapack_int LAPACKE_zungtr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);

Include Files
• mkl.h

Description
The routine explicitly generates the n-by-n unitary matrix Q formed by ?hetrd when reducing a complex
Hermitian matrix A to tridiagonal form: A = Q^*T*Q^H. Use this routine after a call to ?hetrd.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

**uplo**
Must be 'U' or 'L'.
Use the same **uplo** as supplied to **?hetrd**.

**n**
The order of the matrix **Q** (**n** ≥ 0).

**a, tau**
Arrays:
- **a** (size max(1, **lda***(**n**))) is the array **a** as returned by **?hetrd**.
- **tau** is the array **tau** as returned by **?hetrd**.

**lda**
The dimension of **tau** must be at least max(1, **n**-1).
The leading dimension of **a**; at least max(1, **n**).

Output Parameters

**a**
Overwritten by the unitary matrix **Q**.

Return Values

This function returns a value **info**.
If **info**=0, the execution is successful.
If **info** = -i, the i-th parameter had an illegal value.

Application Notes

The computed matrix **Q** differs from an exactly unitary matrix by a matrix **E** such that \( ||E||_2 = O(\varepsilon) \), where \( \varepsilon \) is the machine precision.
The approximate number of floating-point operations is \( (16/3) n^3 \).
The real counterpart of this routine is **orgtr**.

**?unmtr**
*Multiplies a complex matrix by the complex unitary matrix Q determined by ?hetrd.*

Syntax

```c
#include <mkl.h>

lapack_int LAPACKE_cunmtr (int matrix_layout, char side, char uplo, char trans,
                         lapack_int m, lapack_int n, const lapack_complex_float* a,
                         lapack_int lda, const lapack_complex_float* tau,
                         lapack_complex_float* c, lapack_int ldc);

lapack_int LAPACKE_zunmtr (int matrix_layout, char side, char uplo, char trans,
                         lapack_int m, lapack_int n, const lapack_complex_double* a,
                         lapack_int lda, const lapack_complex_double* tau,
                         lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description
The routine multiplies a complex matrix $C$ by $Q$ or $Q^H$, where $Q$ is the unitary matrix $Q$ formed by `?hetrd` when reducing a complex Hermitian matrix $A$ to tridiagonal form: $A = QTQ^H$. Use this routine after a call to `?hetrd`.

Depending on the parameters `side` and `trans`, the routine can form one of the matrix products $Q^C$, $QH^C$, $C^Q$, or $C^QH$ (overwriting the result on $C$).

**Input Parameters**

In the descriptions below, $r$ denotes the order of $Q$:

If `side` = 'L', $r = m$; if `side` = 'R', $r = n$.

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **side**
  Must be either 'L' or 'R'.
  - If `side` = 'L', $Q$ or $Q^H$ is applied to $C$ from the left.
  - If `side` = 'R', $Q$ or $Q^H$ is applied to $C$ from the right.

- **uplo**
  Must be 'U' or 'L'.
  Use the same `uplo` as supplied to `?hetrd`.

- **trans**
  Must be either 'N' or 'T'.
  - If `trans` = 'N', the routine multiplies $C$ by $Q$.
  - If `trans` = 'C', the routine multiplies $C$ by $Q^H$.

- **m**
  The number of rows in the matrix $C$ ($m \geq 0$).

- **n**
  The number of columns in $C$ ($n \geq 0$).

- **a, c, tau**
  $a$ (size max(1, $lda \times r$)) and $tau$ are the arrays returned by `?hetrd`.
  The dimension of $tau$ must be at least max(1, $r-1$).
  $c$(size max(1, $ldc \times n$) for column major layout and max(1, $ldc \times m$) for row major layout) contains the matrix $C$.

- **lda**
  The leading dimension of $a$; $lda \geq \text{max}(1, r)$.

- **ldc**
  The leading dimension of $c$; $ldc \geq \text{max}(1, n)$ for column major layout and $ldc \geq \text{max}(1, m)$ for row major layout.

**Output Parameters**

- **c**
  Overwritten by the product $Q^C$, $Q^H^C$, $C^Q$, or $C^QH$ (as specified by `side` and `trans`).

**Return Values**

This function returns a value `info`.

- If `info`=0, the execution is successful.
- If `info` = $-i$, the $i$-th parameter had an illegal value.
Application Notes

The computed product differs from the exact product by a matrix $E$ such that $||E||_2 = O(\varepsilon) * ||C||_2$, where $\varepsilon$ is the machine precision.

The total number of floating-point operations is approximately $8m^2n$ if $side = 'L'$ or $8n^2m$ if $side = 'R'$.

The real counterpart of this routine is `ormtr`.

`?sptrd`

Reduces a real symmetric matrix to tridiagonal form using packed storage.

Syntax

```c
lapack_int LAPACKE_ssptrd (int matrix_layout, char uplo, lapack_int n, float* ap, float* d, float* e, float* tau);

lapack_int LAPACKE_dsptrd (int matrix_layout, char uplo, lapack_int n, double* ap, double* d, double* e, double* tau);
```

Include Files

- mkl.h

Description

The routine reduces a packed real symmetric matrix $A$ to symmetric tridiagonal form $T$ by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix $Q$ is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with $Q$ in this representation. See Application Notes below for details.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `uplo` Must be 'U' or 'L'.
  - If `uplo` = 'U', `ap` stores the packed upper triangle of $A$.
  - If `uplo` = 'L', `ap` stores the packed lower triangle of $A$.

- `n` The order of the matrix $A$ ($n \geq 0$).

- `ap` Array, size at least max(1, n(n+1)/2). Contains either upper or lower triangle of $A$ (as specified by `uplo`) in the packed form described in Matrix Storage Schemes.

Output Parameters

- `ap` Overwritten by the tridiagonal matrix $T$ and details of the orthogonal matrix $Q$, as specified by `uplo`.

- `d`, `e`, `tau` Arrays:
  - $d$ contains the diagonal elements of the matrix $T$.
  - The dimension of $d$ must be at least max(1, n).
  - $e$ contains the off-diagonal elements of $T$. 

The dimension of \( e \) must be at least \( \max(1, n-1) \).

\( \tau \) Stores \((n-1)\) scalars that define elementary reflectors in decomposition of the matrix \( Q \) in a product of \( n-1 \) reflectors.

The dimension of \( \tau \) must be at least \( \max(1, n-1) \).

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The matrix \( Q \) is represented as a product of \( n-1 \) elementary reflectors, as follows:

- If \( uplo = 'U' \), \( Q = H(n-1) \ldots H(2)H(1) \)
  
  Each \( H(i) \) has the form
  \[
  H(i) = I - \tau *v*v^T
  \]
  
  where \( \tau \) is a real scalar and \( v \) is a real vector with \( v(i+1:n) = 0 \) and \( v(i) = 1 \).
  
  On exit, \( \tau \) is stored in \( \tau[i-1] \), and \( v(1:i-1) \) is stored in \( AP \), overwriting \( A(l:i-1, i+1) \).

- If \( uplo = 'L' \), \( Q = H(1)H(2) \ldots H(n-1) \)
  
  Each \( H(i) \) has the form
  \[
  H(i) = I - \tau *v*v^T
  \]
  
  where \( \tau \) is a real scalar and \( v \) is a real vector with \( v(1:i) = 0 \) and \( v(i+1) = 1 \).
  
  On exit, \( \tau \) is stored in \( \tau[i-1] \), and \( v(i+2:n) \) is stored in \( AP \), overwriting \( A(i+2:n, i) \).

The computed matrix \( T \) is exactly similar to a matrix \( A+E \), where \( ||E||_2 = \epsilon *||A||_2 \), \( c(n) \) is a modestly increasing function of \( n \), and \( \epsilon \) is the machine precision. The approximate number of floating-point operations is \((4/3)n^3\).

After calling this routine, you can call the following:

- opgtr to form the computed matrix \( Q \) explicitly
- opmtr to multiply a real matrix by \( Q \).

The complex counterpart of this routine is hptrd.

**?opgtr**

Generates the real orthogonal matrix \( Q \) determined by ?sptrd.

**Syntax**

```c
lapack_int LAPACKE_sopgtr (char matrix_layout, char uplo, int n, const float* ap, const float* tau, float* q, int ldq);

lapack_int LAPACKE_dopgtr (char matrix_layout, char uplo, int n, const double* ap, const double* tau, double* q, int ldq);
```
Include Files
• mkl.h

Description
The routine explicitly generates the \( n \)-by-\( n \) orthogonal matrix \( Q \) formed by \( \text{sptrd} \) when reducing a packed real symmetric matrix \( A \) to tridiagonal form: \( A = Q^T T Q \). Use this routine after a call to \( \text{?sptrd} \).

Input Parameters

\( \text{matrix}_\text{layout} \)

Specifies whether matrix storage layout is row major (\text{LAPACK\_ROW\_MAJOR}) or column major (\text{LAPACK\_COL\_MAJOR}).

\( \text{uplo} \)

Must be ‘U’ or ‘L’. Use the same \( \text{uplo} \) as supplied to \( \text{?sptrd} \).

\( n \)

The order of the matrix \( Q \) (\( n \geq 0 \)).

\( \text{ap, tau} \)

Arrays \( \text{ap} \) and \( \text{tau} \), as returned by \( \text{?sptrd} \).

The size of \( \text{ap} \) must be at least \( \text{max}(1, n(n+1)/2) \).

The size of \( \text{tau} \) must be at least \( \text{max}(1, n-1) \).

\( \text{ldq} \)

The leading dimension of the output array \( q \); at least \( \text{max}(1, n) \).

Output Parameters

\( q \)

Array, size (size \( \text{max}(1, \text{ldq} \times n) \)).

Contains the computed matrix \( Q \).

Return Values
This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes
The computed matrix \( Q \) differs from an exactly orthogonal matrix by a matrix \( E \) such that \( \|E\|_2 = O(\epsilon) \), where \( \epsilon \) is the machine precision.

The approximate number of floating-point operations is \( (4/3) n^3 \).

The complex counterpart of this routine is \( \text{upgtr} \).

\( \text{?opmtr} \)

\text{Multiplies a real matrix by the real orthogonal matrix} \ Q \ \text{determined by} \ ?\text{sptrd}.

Syntax
lapack_int \text{LAPACKE\_sopmtr} (int \text{matrix\_layout}, char \text{side}, char \text{uplo}, char \text{trans},
lapack_int \text{m}, lapack_int \text{n}, const float* \text{ap}, const float* \text{tau}, float* \text{c}, lapack_int \text{ldc});

lapack_int \text{LAPACKE\_dopmtr} (int \text{matrix\_layout}, char \text{side}, char \text{uplo}, char \text{trans},
lapack_int \text{m}, lapack_int \text{n}, const double* \text{ap}, const double* \text{tau}, double* \text{c}, lapack_int \text{ldc});
Include Files

- mkl.h

Description

The routine multiplies a real matrix $C$ by $Q$ or $Q^T$, where $Q$ is the orthogonal matrix formed by sptrd when reducing a packed real symmetric matrix $A$ to tridiagonal form: $A = Q^* T Q$. Use this routine after a call to ?sptrd.

Depending on the parameters side and trans, the routine can form one of the matrix products $Q^* C$, $Q^T C$, $C^* Q$, or $C^* Q^T$ (overwriting the result on $C$).

Input Parameters

In the descriptions below, $r$ denotes the order of $Q$:

If side = 'L', $r = m$; if side = 'R', $r = n$.

- matrix_layout
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- side
  Must be either 'L' or 'R'.
  - If side = 'L', $Q$ or $Q^T$ is applied to $C$ from the left.
  - If side = 'R', $Q$ or $Q^T$ is applied to $C$ from the right.

- uplo
  Must be 'U' or 'L'.
  Use the same uplo as supplied to ?sptrd.

- trans
  Must be either 'N' or 'T'.
  - If trans = 'N', the routine multiplies $C$ by $Q$.
  - If trans = 'T', the routine multiplies $C$ by $Q^T$.

- m
  The number of rows in the matrix $C$ ($m \geq 0$).

- n
  The number of columns in $C$ ($n \geq 0$).

- ap, tau, c
  $ap$ and $tau$ are the arrays returned by ?sptrd.
  The dimension of $ap$ must be at least max(1, $r(r+1)/2$).
  The dimension of $tau$ must be at least max(1, $r-1$).
  $c$(size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$) for row major layout) contains the matrix $C$.

- ldc
  The leading dimension of $c$; $ldc \geq$ max(1, $n$) for column major layout and $ldc \geq$ max(1, $m$) for row major layout.

Output Parameters

- c
  Overwritten by the product $Q^* C$, $Q^T C$, $C^* Q$, or $C^* Q^T$ (as specified by $side$ and $trans$).

Return Values

This function returns a value info.
If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

The computed product differs from the exact product by a matrix $E$ such that $||E||_2 = O(\varepsilon) ||C||_2$, where $\varepsilon$ is the machine precision.

The total number of floating-point operations is approximately $2^*m^2*n$ if $side = 'L'$, or $2^*n^2*m$ if $side = 'R'$.

The complex counterpart of this routine is upmtr.

**?hprrd**

Reduces a complex Hermitian matrix to tridiagonal form using packed storage.

**Syntax**

```c
lapack_int LAPACKE_chptrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* ap, float* d, float* e, lapack_complex_float* tau );
```

```c
lapack_int LAPACKE_zhptrd( int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* ap, double* d, double* e, lapack_complex_double* tau );
```

**Include Files**

- mkl.h

**Description**

The routine reduces a packed complex Hermitian matrix $A$ to symmetric tridiagonal form $T$ by a unitary similarity transformation: $A = Q*T*Q^H$. The unitary matrix $Q$ is not formed explicitly but is represented as a product of $n-1$ elementary reflectors. Routines are provided for working with $Q$ in this representation (see Application Notes below).

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  
  Must be 'U' or 'L'.

  - If $uplo = 'U'$, $ap$ stores the packed upper triangle of $A$.
  - If $uplo = 'L'$, $ap$ stores the packed lower triangle of $A$.

- **n**

  The order of the matrix $A$ ($n \geq 0$).

- **ap**

  Array, size at least max(1, $n(n+1)/2$). Contains either upper or lower triangle of $A$ (as specified by $uplo$) in the packed form described in Matrix Storage Schemes.

**Output Parameters**

- **ap**

  Overwritten by the tridiagonal matrix $T$ and details of the unitary matrix $Q$, as specified by $uplo$.

- **d, e**

  Arrays:
\(d\) contains the diagonal elements of the matrix \(T\).
The size of \(d\) must be at least \(\max(1, n)\).

\(e\) contains the off-diagonal elements of \(T\).
The size of \(e\) must be at least \(\max(1, n-1)\).

\(\text{tau}\)
Array, size at least \(\max(1, n-1)\). Stores \((n-1)\) scalars that define elementary reflectors in decomposition of the unitary matrix \(Q\) in a product of reflectors.

**Return Values**
This function returns a value \(\text{info}\).

If \(\text{info}=0\), the execution is successful.

If \(\text{info} = -i\), the \(i\)-th parameter had an illegal value.

**Application Notes**
The computed matrix \(T\) is exactly similar to a matrix \(A + E\), where \(||E||_2 = c(n) \times \varepsilon \times ||A||_2\), \(c(n)\) is a modestly increasing function of \(n\), and \(\varepsilon\) is the machine precision.

The approximate number of floating-point operations is \((16/3) n^3\).

After calling this routine, you can call the following:

\text{upgtr} to form the computed matrix \(Q\) explicitly
\text{upmtr} to multiply a complex matrix by \(Q\).

The real counterpart of this routine is \text{sptrd}.

**?upgtr**
*Generates the complex unitary matrix \(Q\) determined by ?hptrd.*

**Syntax**

```c
lapack_int LAPACKE_cupgtr (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_float* ap, const lapack_complex_float* tau, lapack_complex_float* q,
lapack_int ldq);
```

```c
lapack_int LAPACKE_zupgtr (int matrix_layout, char uplo, lapack_int n, const
lapack_complex_double* ap, const lapack_complex_double* tau, lapack_complex_double* q,
lapack_int ldq);
```

**Include Files**

- \text{mkl.h}

**Description**
The routine explicitly generates the \(n\)-by-\(n\) unitary matrix \(Q\) formed by \text{hptrd} when reducing a packed complex Hermitian matrix \(A\) to tridiagonal form: \(A = Q^* T \times Q^H\). Use this routine after a call to ?hptrd.
Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'. Use the same uplo as supplied to ?hptrd.
- **n**: The order of the matrix Q (n ≥ 0).
- **ap, tau**: Arrays ap and tau, as returned by ?hptrd. The dimension of ap must be at least max(1, n(n+1)/2). The dimension of tau must be at least max(1, n-1).
- **ldq**: The leading dimension of the output array q; at least max(1, n).

Output Parameters

- **q**: Array, size (size max(1, ldq*n)). Contains the computed matrix Q.

Return Values

This function returns a value info.
- If info=0, the execution is successful.
- If info = -i, the i-th parameter had an illegal value.

Application Notes

The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that \( ||E||_2 = O(\varepsilon) \), where \( \varepsilon \) is the machine precision.

The approximate number of floating-point operations is \( (16/3)n^3 \).

The real counterpart of this routine is opgtr.

**?upmtr**

*Multiplies a complex matrix by the unitary matrix Q determined by ?hptrd.*

Syntax

```c
lapack_int LAPACKE_cupmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_float* ap, const lapack_complex_float* tau, lapack_complex_float* c, lapack_int ldc);
```

```c
lapack_int LAPACKE_zupmtr (int matrix_layout, char side, char uplo, char trans,
lapack_int m, lapack_int n, const lapack_complex_double* ap, const
lapack_complex_double* tau, lapack_complex_double* c, lapack_int ldc);
```

Include Files

- mkl.h

Description
The routine multiplies a complex matrix $C$ by $Q$ or $Q^H$, where $Q$ is the unitary matrix formed by hptrd when reducing a packed complex Hermitian matrix $A$ to tridiagonal form: $A = Q^T*Q^H$. Use this routine after a call to ?hptrd.

Depending on the parameters side and trans, the routine can form one of the matrix products $Q*C$, $Q^H*C$, $C*Q$, or $C*Q^H$ (overwriting the result on $C$).

**Input Parameters**

In the descriptions below, $r$ denotes the order of $Q$:

If side = 'L', $r = m$; if side = 'R', $r = n$.

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **side**
  Must be either 'L' or 'R'.
  - If side = 'L', $Q$ or $Q^H$ is applied to $C$ from the left.
  - If side = 'R', $Q$ or $Q^H$ is applied to $C$ from the right.

- **uplo**
  Must be 'U' or 'L'.
  Use the same uplo as supplied to ?hptrd.

- **trans**
  Must be either 'N' or 'T'.
  - If trans = 'N', the routine multiplies $C$ by $Q$.
  - If trans = 'T', the routine multiplies $C$ by $Q^H$.

- **m**
  The number of rows in the matrix $C$ ($m \geq 0$).

- **n**
  The number of columns in $C$ ($n \geq 0$).

- **ap, tau, c,**
  $ap$ and $tau$ are the arrays returned by ?hptrd.
  - The size of $ap$ must be at least max(1, $r(r+1)/2$).
  - The size of $tau$ must be at least max(1, $r-1$).
  - $c$(size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$) for row major layout) contains the matrix $C$.

- **ldc**
  The leading dimension of $c$; $ldc \geq$ max(1, $m$) for column major layout and $ldc \geq$ max(1, $n$) for row major layout.

**Output Parameters**

- **c**
  Overwritten by the product $Q*C$, $Q^H*C$, $C*Q$, or $C*Q^H$ (as specified by side and trans).

**Return Values**

This function returns a value $info$.

- If $info=0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.
Application Notes

The computed product differs from the exact product by a matrix $E$ such that $||E||_2 = O(\varepsilon)*||C||_2$, where $\varepsilon$ is the machine precision.

The total number of floating-point operations is approximately $8*{m}^2*{n}$ if $side = 'L'$ or $8*{n}^2*{m}$ if $side = 'R'$.

The real counterpart of this routine is $\text{opmtr}$.

**sbtrd**

Reduces a real symmetric band matrix to tridiagonal form.

**Syntax**

```c
lapack_int LAPACKE_ssbtrd (int matrix_layout, char vect, char uplo, lapack_int n, lapack_int kd, float* ab, lapack_int ldab, float* d, float* e, float* q, lapack_int ldq);
lapack_int LAPACKE_dsbtrd (int matrix_layout, char vect, char uplo, lapack_int n, lapack_int kd, double* ab, lapack_int ldab, double* d, double* e, double* q, lapack_int ldq);
```

**Include Files**

- mkl.h

**Description**

The routine reduces a real symmetric band matrix $A$ to symmetric tridiagonal form $T$ by an orthogonal similarity transformation: $A = Q*T*Q^T$. The orthogonal matrix $Q$ is determined as a product of Givens rotations.

If required, the routine can also form the matrix $Q$ explicitly.

**Input Parameters**

- $matrix\_layout$ 
  Specifies whether matrix storage layout is row major ($\text{LAPACK\_ROW\_MAJOR}$) or column major ($\text{LAPACK\_COL\_MAJOR}$).

- $vect$ 
  Must be 'V', 'N', or 'U'.
  - If $vect = 'V'$, the routine returns the explicit matrix $Q$.
  - If $vect = 'N$', the routine does not return $Q$.
  - If $vect = 'U'$, the routine updates matrix $X$ by forming $X*Q$.

- $uplo$ 
  Must be 'U' or 'L'.
  - If $uplo = 'U'$, $ab$ stores the upper triangular part of $A$.
  - If $uplo = 'L'$, $ab$ stores the lower triangular part of $A$.

- $n$ 
  The order of the matrix $A$ ($n \geq 0$).

- $kd$ 
  The number of super- or sub-diagonals in $A$ ($kd \geq 0$).
**ab, q**

- **ab**: An array containing either
  - upper or lower triangular part of the matrix A (as specified by `uplo`) in band storage format.
  - 
  - **q**: An array.
    - If `vect` = 'U', the q array must contain an n-by-n matrix X.
    - If `vect` = 'N' or 'V', the q parameter need not be set.

**ldab**

- The leading dimension of ab; at least \( kd + 1 \) for column major layout and \( n \) for row major layout.

**ldq**

- The leading dimension of q. Constraints:
  - \( ldq \geq \max(1, n) \) if `vect` = 'V' or 'U';
  - \( ldq \geq 1 \) if `vect` = 'N'.

**Output Parameters**

- **ab**
  - On exit, the diagonal elements of the array ab are overwritten by the diagonal elements of the tridiagonal matrix T. If \( kd > 0 \), the elements on the first superdiagonal (if `uplo` = 'U') or the first subdiagonal (if `uplo` = 'L') are overwritten by the off-diagonal elements of T. The rest of ab is overwritten by values generated during the reduction.

- **d, e, q**
  - Arrays:
    - \( d \) contains the diagonal elements of the matrix T.
    - The size of d must be at least \( \max(1, n) \).
    - \( e \) contains the off-diagonal elements of T.
    - The size of e must be at least \( \max(1, n-1) \).
    - \( q \) is not referenced if `vect` = 'N'.
    - If `vect` = 'V', q contains the n-by-n matrix Q.
    - If `vect` = 'U', q contains the product X* Q.

**Return Values**

This function returns a value `info`.

- If `info`=0, the execution is successful.
- If `info` = -i, the i-th parameter had an illegal value.

**Application Notes**

The computed matrix T is exactly similar to a matrix \( A+E \), where \( ||E||_2 = c(n) \varepsilon ||A||_2 \), \( c(n) \) is a modestly increasing function of \( n \), and \( \varepsilon \) is the machine precision. The computed matrix Q differs from an exactly orthogonal matrix by a matrix E such that \( ||E||_2 = O(\varepsilon) \).

The total number of floating-point operations is approximately \( 6n^2kd \) if `vect` = 'N', with \( 3n^3(kd-1)/kd \) additional operations if `vect` = 'V'.

The complex counterpart of this routine is `hbtrd`.
?hbtrd
Reduces a complex Hermitian band matrix to tridiagonal form.

Syntax

```c
lapack_int LAPACKE_chbtrd( int matrix_layout, char vect, char uplo, lapack_int n, lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* d, float* e, lapack_complex_float* q, lapack_int ldq );
```

```c
lapack_int LAPACKE_zhbtrd( int matrix_layout, char vect, char uplo, lapack_int n, lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* d, double* e, lapack_complex_double* q, lapack_int ldq );
```

Include Files

- mkl.h

Description

The routine reduces a complex Hermitian band matrix $A$ to symmetric tridiagonal form $T$ by a unitary similarity transformation: $A = Q^*TQ^H$. The unitary matrix $Q$ is determined as a product of Givens rotations. If required, the routine can also form the matrix $Q$ explicitly.

Input Parameters

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **vect**
  
  Must be 'V', 'N', or 'U'.
  
  If $\text{vect} = 'V'$, the routine returns the explicit matrix $Q$.
  
  If $\text{vect} = 'N'$, the routine does not return $Q$.
  
  If $\text{vect} = 'U'$, the routine updates matrix $X$ by forming $Q^*X$.

- **uplo**
  
  Must be 'U' or 'L'.
  
  If $\text{uplo} = 'U'$, $ab$ stores the upper triangular part of $A$.
  
  If $\text{uplo} = 'L'$, $ab$ stores the lower triangular part of $A$.

- **n**
  
  The order of the matrix $A (n \geq 0)$.

- **kd**
  
  The number of super- or sub-diagonals in $A (kd \geq 0)$.

- **ab**
  
  $(size$ least max$(1, ldab*n))$ for column major layout and at least max$(1, ldab*(kd+1))$ for row major layout) is an array containing either upper or lower triangular part of the matrix $A$ (as specified by $\text{uplo}$) in band storage format.

- **q**
  
  $(size max(1, ldq*n))$ is an array.
  
  If $\text{vect} = 'U'$, the $q$ array must contain an $n$-by-$n$ matrix $X$.
  
  If $\text{vect} = 'N'$ or 'V', the $q$ parameter need not be set.'

- **ldab**
  
  The leading dimension of $ab$; at least $kd+1$ for column major layout and $n$ for row major layout.
The leading dimension of `q`. Constraints:

- `ldq ≥ max(1, n)` if `vect = 'V'` or `U`;
- `ldq ≥ 1` if `vect = 'N'`.

**Output Parameters**

- `ab`: On exit, the diagonal elements of the array `ab` are overwritten by the diagonal elements of the tridiagonal matrix `T`. If `kd > 0`, the elements on the first superdiagonal (if `uplo = 'U'`) or the first subdiagonal (if `uplo = 'L'`) are overwritten by the off-diagonal elements of `T`. The rest of `ab` is overwritten by values generated during the reduction.

- `d, e`: Arrays:
  - `d` contains the diagonal elements of the matrix `T`. The dimension of `d` must be at least `max(1, n)`.
  - `e` contains the off-diagonal elements of `T`. The dimension of `e` must be at least `max(1, n-1)`.

- `q`: If `vect = 'N'`, `q` is not referenced.
  - If `vect = 'V'`, `q` contains the `n`-by-`n` matrix `Q`.
  - If `vect = 'U'`, `q` contains the product `X* Q`.

**Return Values**

This function returns a value `info`.

- If `info=0`, the execution is successful.
- If `info = -i`, the `i`-th parameter had an illegal value.

**Application Notes**

The computed matrix `T` is exactly similar to a matrix `A + E`, where `||E||_2 = c(n) * ε * ||A||_2`, `c(n)` is a modestly increasing function of `n`, and `ε` is the machine precision. The computed matrix `Q` differs from an exactly unitary matrix by a matrix `E` such that `||E||_2 = O(ε)`.

The total number of floating-point operations is approximately `20n^2 * kd` if `vect = 'N'`, with `10n^3 * (kd-1) / kd` additional operations if `vect = 'V'`.

The real counterpart of this routine is `sbtrd`.

`?sterf`  
*Computes all eigenvalues of a real symmetric tridiagonal matrix using QR algorithm.*

**Syntax**

```c
lapack_int LAPACKE_ssterf (lapack_int n, float* d, float* e);
lapack_int LAPACKE_dsterf (lapack_int n, double* d, double* e);
```

**Include Files**

- `mkl.h`
Description

The routine computes all the eigenvalues of a real symmetric tridiagonal matrix $T$ (which can be obtained by reducing a symmetric or Hermitian matrix to tridiagonal form). The routine uses a square-root-free variant of the QR algorithm.

If you need not only the eigenvalues but also the eigenvectors, call steqr.

Input Parameters

$n$  
The order of the matrix $T$ ($n \geq 0$).

$d$, $e$  
Arrays:
$d$ contains the diagonal elements of $T$.

The dimension of $d$ must be at least max(1, $n$).

$e$ contains the off-diagonal elements of $T$.

The dimension of $e$ must be at least max(1, $n$-1).

Output Parameters

$d$  
The $n$ eigenvalues in ascending order, unless info $> 0$.

See also info.

$e$  
On exit, the array is overwritten; see info.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info $= i$, the algorithm failed to find all the eigenvalues after 30$n$ iterations:

$i$ off-diagonal elements have not converged to zero. On exit, $d$ and $e$ contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to $T$.

If info $= -i$, the $i$-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+\mathbf{E}$ such that $||\mathbf{E}||_2 = O(\varepsilon) * ||T||_2$, where $\varepsilon$ is the machine precision.

If $\lambda_i$ is an exact eigenvalue, and $m_i$ is the corresponding computed value, then

$|\mu_i - \lambda_i| \leq c(n) * \varepsilon * ||T||_2$

where $c(n)$ is a modestly increasing function of $n$.

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about $14n^2$.

?steqr

Computes all eigenvalues and eigenvectors of a symmetric or Hermitian matrix reduced to tridiagonal form (QR algorithm).
Syntax

```c
lapack_int LAPACKE_ssteqr(int matrix_layout, char compz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dsteqr(int matrix_layout, char compz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_csteqr(int matrix_layout, char compz, lapack_int n, float* d, float* e, lapack_complex_float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_zsteqr(int matrix_layout, char compz, lapack_int n, double* d, double* e, lapack_complex_double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric tridiagonal matrix \( T \). In other words, the routine can compute the spectral factorization: \( T = Z \Lambda Z^T \). Here \( \Lambda \) is a diagonal matrix whose diagonal elements are the eigenvalues \( \lambda_i \); \( Z \) is an orthogonal matrix whose columns are eigenvectors. Thus, \( Tz_i = \lambda_i z_i \) for \( i = 1, 2, \ldots, n \).

The routine normalizes the eigenvectors so that \( ||z_i||_2 = 1 \).

You can also use the routine for computing the eigenvalues and eigenvectors of an arbitrary real symmetric (or complex Hermitian) matrix \( A \) reduced to tridiagonal form \( T \): \( A = Q^*TQ \). In this case, the spectral factorization is as follows: \( A = Q^*TQ = (Q^*\Lambda Q)^H \). Before calling \?steqr, you must reduce \( A \) to tridiagonal form and generate the explicit matrix \( Q \) by calling the following routines:

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>for real matrices:</th>
<th>for complex matrices:</th>
</tr>
</thead>
<tbody>
<tr>
<td>full storage</td>
<td>?sytrd, ?orgtr</td>
<td>?hetrd, ?ungtr</td>
</tr>
<tr>
<td>packed storage</td>
<td>?sptrd, ?opgtr</td>
<td>?hptrd, ?upgtr</td>
</tr>
<tr>
<td>band storage</td>
<td>?sbtrd(vect='V')</td>
<td>?hbtrd(vect='V')</td>
</tr>
</tbody>
</table>

If you need eigenvalues only, it's more efficient to call \?sterf. If \( T \) is positive-definite, \?pteqr can compute small eigenvalues more accurately than \?steqr.

To solve the problem by a single call, use one of the divide and conquer routines \?stevd, \?syevd, \?spevd, or \?sbevd for real symmetric matrices or \?heevd, \?hevd, or \?hbevd for complex Hermitian matrices.

Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **compz**
  - Must be 'N' or 'I' or 'V'.
  - If \( \text{compz} = 'N' \), the routine computes eigenvalues only.
  - If \( \text{compz} = 'I' \), the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix \( T \).
If \( \text{compz} = \text{'V'} \), the routine computes the eigenvalues and eigenvectors of the original symmetric matrix. On entry, \( z \) must contain the orthogonal matrix used to reduce the original matrix to tridiagonal form.

\[ n \]

The order of the matrix \( T \) \((n \geq 0)\).

\[ d, e \]

Arrays:

\( d \) contains the diagonal elements of \( T \).

The size of \( d \) must be at least \( \max(1, n) \).

\( e \) contains the off-diagonal elements of \( T \).

The size of \( e \) must be at least \( \max(1, n-1) \).

\[ z \]

Array, size \( \max(1, ldz \times n) \).

If \( \text{compz} = \text{'N'} \) or \( \text{'I'} \), \( z \) need not be set.

If \( \text{vect} = \text{'V'} \), \( z \) must contain the orthogonal matrix used in the reduction to tridiagonal form.

\[ ldz \]

The leading dimension of \( z \). Constraints:

\( ldz \geq 1 \) if \( \text{compz} = \text{'N'} \);

\( ldz \geq \max(1, n) \) if \( \text{compz} = \text{'V'} \) or \( \text{'I'} \).

**Output Parameters**

\[ d \]

The \( n \) eigenvalues in ascending order, unless \( \text{info} > 0 \).

See also \( \text{info} \).

\[ e \]

On exit, the array is overwritten; see \( \text{info} \).

\[ z \]

If \( \text{info} = 0 \), contains the \( n \)-by-\( n \) matrix the columns of which are orthonormal eigenvectors (the \( i \)-th column corresponds to the \( i \)-th eigenvalue).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = i \), the algorithm failed to find all the eigenvalues after \( 30n \) iterations: \( i \) off-diagonal elements have not converged to zero. On exit, \( d \) and \( e \) contain, respectively, the diagonal and off-diagonal elements of a tridiagonal matrix orthogonally similar to \( T \).

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The computed eigenvalues and eigenvectors are exact for a matrix \( T+E \) such that \( \| E \|_2 = O(\varepsilon) \| T \|_2 \), where \( \varepsilon \) is the machine precision.

If \( \lambda_i \) is an exact eigenvalue, and \( \mu_i \) is the corresponding computed value, then

\[ |\mu_i - \lambda_i| \leq c(n) \| \varepsilon \|_2 \| T \|_2 \]

where \( c(n) \) is a modestly increasing function of \( n \).
If $z_i$ is the corresponding exact eigenvector, and $w_i$ is the corresponding computed vector, then the angle $\theta(z_i, w_i)$ between them is bounded as follows:

$$\theta(z_i, w_i) \leq c(n) \varepsilon^* \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.$$ 

The total number of floating-point operations depends on how rapidly the algorithm converges. Typically, it is about

- $24n^2$ if $\text{compz} = 'N'$;
- $7n^3$ (for complex flavors, $14n^3$) if $\text{compz} = 'V$ or 'I'.

?stemr

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

**Syntax**

```c
lapack_int LAPACKE_sstemr( int matrix_layout, char jobz, char range, lapack_int n,
                         const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu,
                         lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
                         lapack_logical* tryrac);
```

```c
lapack_int LAPACKE(dstemr( int matrix_layout, char jobz, char range, lapack_int n,
                         const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
                         lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
                         lapack_logical* tryrac);
```

```c
lapack_int LAPACKE_cstemr( int matrix_layout, char jobz, char range, lapack_int n,
                         const float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu,
                         lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
                         lapack_logical* tryrac);
```

```c
lapack_int LAPACKE_zstemr( int matrix_layout, char jobz, char range, lapack_int n,
                         const double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu,
                         lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int nzc, lapack_int* isuppz,
                         lapack_logical* tryrac);
```

**Include Files**

- mkl.h

**Description**

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix $T$. Any such unreduced matrix has a well defined set of pairwise different real eigenvalues, the corresponding real eigenvectors are pairwise orthogonal.

The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a range of indices $il:iu$ for the desired eigenvalues.

Depending on the number of desired eigenvalues, these are computed either by bisection or the $dqds$ algorithm. Numerically orthogonal eigenvectors are computed by the use of various suitable $L*D*L^T$ factorizations near clusters of close eigenvalues (referred to as RRRs, Relatively Robust Representations). An informal sketch of the algorithm follows.

For each unreduced block (submatrix) of $T$,
a. Compute $T - \sigma \cdot I = L \cdot D \cdot L^T$, so that $L$ and $D$ define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of $L$ and $D$ cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix $T$ does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see steps c and d.

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to step c for any clusters that remain.

Normal execution of ?stemr may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see: [Dhillon04], [Dhillon04-02], [Dhillon97]

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz
Must be 'N' or 'V'.
If jobz = 'N', then only eigenvalues are computed.
If jobz = 'V', then eigenvalues and eigenvectors are computed.

range
Must be 'A' or 'V' or 'I'.
If range = 'A', the routine computes all eigenvalues.
If range = 'V', the routine computes all eigenvalues in the half-open interval: $[vl, vu]$.
If range = 'I', the routine computes eigenvalues with indices $il$ to $iu$.

n
The order of the matrix $T (n \geq 0)$.

d
Array, size $n$.
Contains $n$ diagonal elements of the tridiagonal matrix $T$.

e
Array, size $n$.
Contains $(n-1)$ off-diagonal elements of the tridiagonal matrix $T$ in elements 0 to $n-2$ of e. $e[n-1]$ need not be set on input, but is used internally as workspace.

vl, vu
If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues. Constraint: $vl < vu$.
If range = 'A' or 'I', vl and vu are not referenced.

il, iu
If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$.
If range = 'A' or 'V', il and iu are not referenced.

ldz
The leading dimension of the output array z.
if \( \text{jobz} = 'V' \), then \( \text{idz} \geq \max(1, n) \) for column major layout and \( \text{ldz} \geq \max(1, m) \) for row major layout;

\( \text{idz} \geq 1 \) otherwise.

\( \text{nzc} \)

The number of eigenvectors to be held in the array \( z \).

- If \( \text{range} = 'A' \), then \( \text{nzc} \geq \max(1, n) \);
- If \( \text{range} = 'V' \), then \( \text{nzc} \) is greater than or equal to the number of eigenvalues in the half-open interval: \( (\text{vl}, \text{vu}] \).
- If \( \text{range} = 'I' \), then \( \text{nzc} \geq \text{iu}-\text{il}+1 \).

If \( \text{nzc} = -1 \), then a workspace query is assumed; the routine calculates the number of columns of the array \( z \) that are needed to hold the eigenvectors.

This value is returned as the first entry of the array \( z \), and no error message related to \( \text{nzc} \) is issued by the routine \( \text{xerbla} \).

\( \text{tryrac} \)

If \( \text{tryrac} \) is true, it indicates that the code should check whether the tridiagonal matrix defines its eigenvalues to high relative accuracy. If so, the code uses relative-accuracy preserving algorithms that might be (a bit) slower depending on the matrix. If the matrix does not define its eigenvalues to high relative accuracy, the code can use possibly faster algorithms.

If \( \text{tryrac} \) is not true, the code is not required to guarantee relatively accurate eigenvalues and can use the fastest possible techniques.

**Output Parameters**

\( d \)

On exit, the array \( d \) is overwritten.

\( e \)

On exit, the array \( e \) is overwritten.

\( m \)

The total number of eigenvalues found, \( 0 \leq m \leq n \).

- If \( \text{range} = 'A' \), then \( m=n \), and if \( \text{range} = 'I' \), then \( m=\text{iu}-\text{il}+1 \).

\( w \)

Array, size \( n \).

The first \( m \) elements contain the selected eigenvalues in ascending order.

\( z \)

Array \( z \) (size \( \max(1, \text{ldz} \times m) \) for column major layout and \( \max(1, \text{ldz} \times n) \) for row major layout).

- If \( \text{jobz} = 'V' \), and \( \text{info} = 0 \), then the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( T \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w(i) \).
- If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

Note: the exact value of \( m \) is not known in advance and can be computed with a workspace query by setting \( \text{nzc} = -1 \), see description of the parameter \( \text{nzc} \).

\( \text{isuppz} \)

Array, size \( 2 \times \max(1, m) \).
The support of the eigenvectors in \( z \), that is the indices indicating the nonzero elements in \( z \). The \( i \)-th computed eigenvector is nonzero only in elements \( \text{isuppz}[2*i - 2] \) through \( \text{isuppz}[2*i - 1] \). This is relevant in the case when the matrix is split. \( \text{isuppz} \) is only accessed when \( \text{jobz} = 'V' \) and \( n > 0 \).

\[ \text{tryrac} \]

On exit, \( \text{tryrac} \) is set to true. \( \text{tryrac} \) is set to false if the matrix does not define its eigenvalues to high relative accuracy.

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.
- If \( \text{info} > 0 \), an internal error occurred.

**?stedc**

*Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method.*

**Syntax**

\[
\begin{align*}
\text{lapack_int} & \text{ LAPACKE_sstedc( int matrix_layout, char compz, lapack_int n, float* d, float* e, float* z, lapack_int ldz );} \\
\text{lapack_int} & \text{ LAPACKE_dstedc( int matrix_layout, char compz, lapack_int n, double* d, double* e, double* z, lapack_int ldz );} \\
\text{lapack_int} & \text{ LAPACKE_cstedc( int matrix_layout, char compz, lapack_int n, float* d, float* e, lapack_complex_float* z, lapack_int ldz );} \\
\text{lapack_int} & \text{ LAPACKE_zstedc( int matrix_layout, char compz, lapack_int n, double* d, double* e, lapack_complex_double* z, lapack_int ldz );}
\end{align*}
\]

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a symmetric tridiagonal matrix using the divide and conquer method. The eigenvectors of a full or band real symmetric or complex Hermitian matrix can also be found if \text{sytrd/hetrd} or \text{sptrd/hptrd} or \text{sbtrd/hbtrd} has been used to reduce this matrix to tridiagonal form.

**Input Parameters**

- \( \text{matrix_layout} \) specifies whether matrix storage layout is row major (\text{LAPACK_ROW_MAJOR}) or column major (\text{LAPACK_COL_MAJOR}).
- \( \text{compz} \) must be \( 'N' \) or \( 'I' \) or \( 'V' \).
  - If \( \text{compz} = 'N' \), the routine computes eigenvalues only.
  - If \( \text{compz} = 'I' \), the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix.
If \( \text{compz} = 'V' \), the routine computes the eigenvalues and eigenvectors of the original symmetric/Hermitian matrix. On entry, the array \( z \) must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

\( n \)

The order of the symmetric tridiagonal matrix (\( n \geq 0 \)).

\( d, e \)

Arrays:

\( d \) contains the diagonal elements of the tridiagonal matrix.

The dimension of \( d \) must be at least \( \max(1, n) \).

\( e \) contains the subdiagonal elements of the tridiagonal matrix.

The dimension of \( e \) must be at least \( \max(1, n-1) \).

\( z \)

Array \( z \) is of size \( \max(1, ldz*n) \).

If \( \text{compz} = 'V' \), then, on entry, \( z \) must contain the orthogonal/unitary matrix used to reduce the original matrix to tridiagonal form.

\( ldz \)

The leading dimension of \( z \). Constraints:

\( ldz \geq 1 \) if \( \text{compz} = 'N' \);

\( ldz \geq \max(1, n) \) if \( \text{compz} = 'V' \) or \( 'I' \).

**Output Parameters**

\( d \)

The \( n \) eigenvalues in ascending order, unless \( \text{info} \neq 0 \).

See also \( \text{info} \).

\( e \)

On exit, the array is overwritten; see \( \text{info} \).

\( z \)

If \( \text{info} = 0 \), then if \( \text{compz} = 'V' \), \( z \) contains the orthonormal eigenvectors of the original symmetric/Hermitian matrix, and if \( \text{compz} = 'I' \), \( z \) contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If \( \text{compz} = 'N' \), \( z \) is not referenced.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns \( ij/(n+1) \) through \( \text{mod}(i, n+1) \).

**stegr**

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

**Syntax**

```c
lapack_int LAPACKE_sstegr( int matrix_layout, char jobz, char range, lapack_int n, float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz );
```
lapack_int LAPACKE_dstegr( int matrix_layout, char jobz, char range, lapack_int n,
   double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
   lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_cstegr( int matrix_layout, char jobz, char range, lapack_int n,
   float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
   lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* isuppz );

lapack_int LAPACKE_zstegr( int matrix_layout, char jobz, char range, lapack_int n,
   double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
   lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int* isuppz );

Include Files
• mkl.h

Description
The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal
matrix $T$.
The spectrum may be computed either completely or partially by specifying either an interval $(vl, vu]$ or a
range of indices $il:iu$ for the desired eigenvalues.

stegr is a compatibility wrapper around the improved stemr routine. See its description for further details.
Note that the abstol parameter no longer provides any benefit and hence is no longer used.

Input Parameters

matrix_layout
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz
  Must be 'N' or 'V'.
  If jobz = 'N', then only eigenvalues are computed.
  If jobz = 'V', then eigenvalues and eigenvectors are computed.

range
  Must be 'A' or 'V' or 'I'.
  If range = 'A', the routine computes all eigenvalues.
  If range = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:
  $vl < w[i] \leq vu$.
  If range = 'I', the routine computes eigenvalues with indices $il$ to $iu$.

n
  The order of the matrix $T$ ($n \geq 0$).

d, e
  Arrays:
  $d$ contains the diagonal elements of $T$.
  The dimension of $d$ must be at least max(1, $n$).
  $e$ contains the subdiagonal elements of $T$ in elements 1 to $n-1$; $e(n)$ need not be set on input, but it is used as a workspace.
  The dimension of $e$ must be at least max(1, $n$).
If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: vl < vu.
If range = 'A' or 'I', vl and vu are not referenced.

If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: 1 ≤ il ≤ iu ≤ n, if n > 0.
If range = 'A' or 'V', il and iu are not referenced.

Unused. Was the absolute error tolerance for the eigenvalues/eigenvectors in previous versions.

The leading dimension of the output array z. Constraints:
ldz ≥ 1 if jobz = 'N';
ldz ≥ max(1, n) if jobz = 'V'.

On exit, d and e are overwritten.
The total number of eigenvalues found,
0 ≤ m ≤ n.
If range = 'A', m = n, and if range = 'I', m = iu-il+1.

Array, size at least max(1, n).
The selected eigenvalues in ascending order, stored in w[0] to w[m - 1].

Array z(size max(1, ldz*m)).
If jobz = 'V', and if info = 0, the first m columns of z contain the orthonormal eigenvectors of the matrix T corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with w[i - 1].
If jobz = 'N', then z is not referenced.
Note: if range = 'V', the exact value of m is not known in advance and an upper bound must be used. Using n = m is always safe.

Array, size at least (2*max(1, m)).
The support of the eigenvectors in z, that is the indices indicating the nonzero elements in z. The i-th computed eigenvector is nonzero only in elements isuppz[2*i - 2] through isuppz[2*i - 1]. This is relevant in the case when the matrix is split. isuppz is only accessed when jobz = 'V', and n > 0.

This function returns a value info.
If info=0, the execution is successful.
If $info = -i$, the $i$-th parameter had an illegal value.

If $info > 0$, an internal error occurred.

?pteqr

*Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric positive-definite tridiagonal matrix.*

**Syntax**

```c
lapack_int LAPACKE_spteqr(int matrix_layout, char compz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);
lapack_int LAPACKE_dpteqr(int matrix_layout, char compz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
lapack_int LAPACKE_cpteqr(int matrix_layout, char compz, lapack_int n, float* d, float* e, lapack_complex_float* z, lapack_int ldz);
lapack_int LAPACKE_zpteqr(int matrix_layout, char compz, lapack_int n, double* d, double* e, lapack_complex_double* z, lapack_int ldz);
```

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues and (optionally) all the eigenvectors of a real symmetric positive-definite tridiagonal matrix $T$. In other words, the routine can compute the spectral factorization: $T = Z \Lambda Z^T$.

Here $\Lambda$ is a diagonal matrix whose diagonal elements are the eigenvalues $\lambda_i$; $Z$ is an orthogonal matrix whose columns are eigenvectors. Thus,

$$Tz_i = \lambda_i z_i \text{ for } i = 1, 2, \ldots, n.$$  

(The routine normalizes the eigenvectors so that $||z_i||_2 = 1$.)

You can also use the routine for computing the eigenvalues and eigenvectors of real symmetric (or complex Hermitian) positive-definite matrices $A$ reduced to tridiagonal form $T$: $A = Q^*TQ^H$. In this case, the spectral factorization is as follows: $A = Q^*TQ^H = (QZ)^*\Lambda^* (QZ)^H$. Before calling ?pteqr, you must reduce $A$ to tridiagonal form and generate the explicit matrix $Q$ by calling the following routines:

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>For Real Matrices</th>
<th>For Complex Matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full Storage</strong></td>
<td>?sytrd, ?orgtr</td>
<td>?hetrd, ?ungtr</td>
</tr>
<tr>
<td><strong>Packed Storage</strong></td>
<td>?sptrd, ?opgtr</td>
<td>?hptrd, ?upgtr</td>
</tr>
<tr>
<td><strong>Band Storage</strong></td>
<td>?sbtrd(\text{vec}=\text{'V'})</td>
<td>?hbtrd(\text{vec}=\text{'V'})</td>
</tr>
</tbody>
</table>

The routine first factorizes $T$ as $L^*D^*L^H$ where $L$ is a unit lower bidiagonal matrix, and $D$ is a diagonal matrix. Then it forms the bidiagonal matrix $B = L^1/2$ and calls ?bdsqr to compute the singular values of $B$, which are the square roots of the eigenvalues of $T$. 

827
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

**compz**
Must be 'N' or 'I' or 'V'.
If `compz = 'N'`, the routine computes eigenvalues only.
If `compz = 'I'`, the routine computes the eigenvalues and eigenvectors of the tridiagonal matrix $T$.
If `compz = 'V'`, the routine computes the eigenvalues and eigenvectors of $A$ (and the array $z$ must contain the matrix $Q$ on entry).

**n**
The order of the matrix $T$ ($n \geq 0$).

**d, e**
Arrays:
$d$ contains the diagonal elements of $T$.
The size of $d$ must be at least $\max(1, n)$.
$e$ contains the off-diagonal elements of $T$.
The size of $e$ must be at least $\max(1, n-1)$.

**z**
Array, size $\max(1, ldz*\text{n})$
If `compz = 'N'` or 'I', $z$ need not be set.
If `compz = 'V'`, $z$ must contain the orthogonal matrix used in the reduction to tridiagonal form..

**ldz**
The leading dimension of $z$. Constraints:
$ldz \geq 1$ if `compz = 'N';`
$ldz \geq \max(1, n)$ if `compz = 'V'` or 'I'.

Output Parameters

**d**
The $n$ eigenvalues in descending order, unless $\text{info} > 0$.
See also $\text{info}$.

**e**
On exit, the array is overwritten.

**z**
If $\text{info} = 0$, contains an $n$-by-$n$ matrix the columns of which are orthonormal eigenvectors. (The $i$-th column corresponds to the $i$-th eigenvalue.)

Return Values

This function returns a value $\text{info}$.
If $\text{info}=0$, the execution is successful.
If $\text{info} = i$, the leading minor of order $i$ (and hence $T$ itself) is not positive-definite.
If $\text{info} = n + i$, the algorithm for computing singular values failed to converge; $i$ off-diagonal elements have not converged to zero.
If $\text{info} = -i$, the $i$-th parameter had an illegal value.
Application Notes

If \( \lambda_i \) is an exact eigenvalue, and \( \mu_i \) is the corresponding computed value, then

\[
|\mu_i - \lambda_i| \leq c(n) \varepsilon K^* \lambda_i
\]

where \( c(n) \) is a modestly increasing function of \( n \), \( \varepsilon \) is the machine precision, and

\[
K = \left( \| DTD \|_2 \right)^{1/2}
\]

If \( z_i \) is the corresponding exact eigenvector, and \( w_i \) is the corresponding computed vector, then the angle \( \theta(z_i, w_i) \) between them is bounded as follows:

\[
\theta(z_i, w_i) \leq c(n) \varepsilon K / \min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|). 
\]

Here \( \min_{i \neq j}(|\lambda_i - \lambda_j|/|\lambda_i + \lambda_j|) \) is the relative gap between \( \lambda_i \) and the other eigenvalues.

The total number of floating-point operations depends on how rapidly the algorithm converges.

Typically, it is about

- \( 30n^2 \) if \( \text{compz} = 'N' \);
- \( 6n^3 \) (for complex flavors, \( 12n^3 \)) if \( \text{compz} = 'V' \) or \( 'I' \).

?stebz

Computes selected eigenvalues of a real symmetric tridiagonal matrix by bisection.

Syntax

```c
lapack_int LAPACKE_sstebz (char range, char order, lapack_int n, float vl, float vu,
    lapack_int il, lapack_int iu, float abstol, const float* d, const float* e,
    lapack_int* m, lapack_int* nsplit, float* w, lapack_int* iblock, lapack_int* isplit);

lapack_int LAPACKE_dstebz (char range, char order, lapack_int n, double vl, double vu,
    lapack_int il, lapack_int iu, double abstol, const double* d, const double* e,
    lapack_int* m, lapack_int* nsplit, double* w, lapack_int* iblock, lapack_int* isplit);
```

Include Files

- mkl.h

Description

The routine computes some (or all) of the eigenvalues of a real symmetric tridiagonal matrix \( T \) by bisection. The routine searches for zero or negligible off-diagonal elements to see if \( T \) splits into block-diagonal form \( T = \text{diag}(T_1, T_2, \ldots) \). Then it performs bisection on each of the blocks \( T_i \) and returns the block index of each computed eigenvalue, so that a subsequent call to \( \text{stein} \) can also take advantage of the block structure.

Input Parameters

- **range**
  - Must be 'A' or 'V' or 'I'.
  - If \( \text{range} = 'A' \), the routine computes all eigenvalues.
  - If \( \text{range} = 'V' \), the routine computes eigenvalues \( w[i] \) in the half-open interval: \( vl < w[i] \leq vu \).
  - If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).

- **order**
  - Must be 'B' or 'E'.
If order = 'B', the eigenvalues are to be ordered from smallest to largest within each split-off block.

If order = 'E', the eigenvalues for the entire matrix are to be ordered from smallest to largest.

$n$

The order of the matrix $T$ ($n \geq 0$).

$vl, vu$

If range = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:

$vl < w[i]) \leq vu$.

If range = 'A' or 'I', $vl$ and $vu$ are not referenced.

$il, iu$

Constraint: $1 \leq il \leq iu \leq n$.

If range = 'I', the routine computes eigenvalues $w[i]$ such that $il \leq i \leq iu$ (assuming that the eigenvalues $w[i]$ are in ascending order).

If range = 'A' or 'V', $il$ and $iu$ are not referenced.

$abstol$

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width $abstol$.

If $abstol \leq 0.0$, then the tolerance is taken as $\text{eps} \cdot |T|$, where $\text{eps}$ is the machine precision, and $|T|$ is the 1-norm of the matrix $T$.

$d, e$

Arrays:

$d$ contains the diagonal elements of $T$.

The size of $d$ must be at least $\max(1, n)$.

$e$ contains the off-diagonal elements of $T$.

The size of $e$ must be at least $\max(1, n-1)$.

Output Parameters

$m$

The actual number of eigenvalues found.

$nsplit$

The number of diagonal blocks detected in $T$.

$w$

Array, size at least $\max(1, n)$. The computed eigenvalues, stored in $w[0]$ to $w[m - 1]$.

$iblock, isplit$

Arrays, size at least $\max(1, n)$.

A positive value $iblock[i]$ is the block number of the eigenvalue stored in $w[i]$ (see also info).

The leading $nsplit$ elements of $isplit$ contain points at which $T$ splits into blocks $T_j$ as follows: the block $T_1$ contains rows/columns 1 to $isplit[0]$; the block $T_2$ contains rows/columns $isplit[0]+1$ to $isplit[1]$, and so on.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.
If $info = 1$, for $range = 'A'$ or 'V', the algorithm failed to compute some of the required eigenvalues to the desired accuracy; $iblock[i] < 0$ indicates that the eigenvalue stored in $w[i]$ failed to converge.

If $info = 2$, for $range = 'I'$, the algorithm failed to compute some of the required eigenvalues. Try calling the routine again with $range = 'A'$.

If $info = 3$:
- for $range = 'A'$ or 'V', same as $info = 1$;
- for $range = 'I'$, same as $info = 2$.

If $info = 4$, no eigenvalues have been computed. The floating-point arithmetic on the computer is not behaving as expected.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

The eigenvalues of $T$ are computed to high relative accuracy which means that if they vary widely in magnitude, then any small eigenvalues will be computed more accurately than, for example, with the standard QR method. However, the reduction to tridiagonal form (prior to calling the routine) may exclude the possibility of obtaining high relative accuracy in the small eigenvalues of the original matrix if its eigenvalues vary widely in magnitude.

?stein

Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix.

Syntax

```c
lapack_int LAPACKE_sstein( int matrix_layout, lapack_int n, const float* d, const float* e, lapack_int m, const float* w, const lapack_int* iblock, const lapack_int* isplit, float* z, lapack_int ldz, lapack_int* ifailv);
```

```c
lapack_int LAPACKE_dstein( int matrix_layout, lapack_int n, const double* d, const double* e, lapack_int m, const double* w, const lapack_int* iblock, const lapack_int* isplit, double* z, lapack_int ldz, lapack_int* ifailv);
```

```c
lapack_int LAPACKE_cstein( int matrix_layout, lapack_int n, const float* d, const float* e, lapack_int m, const float* w, const lapack_int* iblock, const lapack_int* isplit, lapack_complex_float* z, lapack_int ldz, lapack_int* ifailv);
```

```c
lapack_int LAPACKE_zstein( int matrix_layout, lapack_int n, const double* d, const double* e, lapack_int m, const double* w, const lapack_int* iblock, const lapack_int* isplit, lapack_complex_double* z, lapack_int ldz, lapack_int* ifailv);
```

Include Files

- mkl.h

Description

The routine computes the eigenvectors of a real symmetric tridiagonal matrix $T$ corresponding to specified eigenvalues, by inverse iteration. It is designed to be used in particular after the specified eigenvalues have been computed by ?stebz with $order = 'B'$, but may also be used when the eigenvalues have been computed by other routines.

If you use this routine after ?stebz, it can take advantage of the block structure by performing inverse iteration on each block $T_i$ separately, which is more efficient than using the whole matrix $T$. 
If $T$ has been formed by reduction of a full symmetric or Hermitian matrix $A$ to tridiagonal form, you can transform eigenvectors of $T$ to eigenvectors of $A$ by calling ?ormtr or ?opmtr (for real flavors) or by calling ?unmtr or ?upmtr (for complex flavors).

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **$n$**: The order of the matrix $T$ ($n \geq 0$).
- **$m$**: The number of eigenvectors to be returned.
- **$d$, $e$, $w$**: Arrays:
  - $d$ contains the diagonal elements of $T$.
  - $e$ contains the sub-diagonal elements of $T$ stored in elements 1 to $n-1$.
  - $w$ contains the eigenvalues of $T$, stored in $w[0]$ to $w[m-1]$ (as returned by stebz). Eigenvalues of $T_1$ must be supplied first, in non-decreasing order; then those of $T_2$, again in non-decreasing order, and so on. Constraint:
    
    $\text{if } \text{iblock}[i] = \text{iblock}[i+1], w[i] \leq w[i+1]$. The size of $w$ must be at least max(1, $n$).

- **$\text{iblock}, \text{isplit}$**: Arrays, size at least max(1, $n$). The arrays $\text{iblock}$ and $\text{isplit}$, as returned by ?stebz with order = 'B'.
  - If you did not call ?stebz with order = 'B', set all elements of $\text{iblock}$ to 1, and $\text{isplit}[0]$ to $n$.

- **$ldz$**: The leading dimension of the output array $z$; $ldz \geq \text{max}(1, n)$ for column major layout and $ldz = \text{max}(1, m)$ for row major layout.

**Output Parameters**

- **$z$**: Array, size at least max(1, $ldz*m$) for column major layout and max(1, $ldz*n$) for row major layout.
  - If $info = 0$, $z$ contains an $n$-by-$n$ matrix the columns of which are orthonormal eigenvectors. (The $i$-th column corresponds to the $i$th eigenvalue.)

- **$\text{ifailv}$**: Array, size at least max(1, $m$).
  - If $info = i > 0$, the first $i$ elements of $\text{ifailv}$ contain the indices of any eigenvectors that failed to converge.

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.
If $info = i$, then $i$ eigenvectors (as indicated by the parameter $ifailv$) each failed to converge in 5 iterations. The current iterates are stored in the corresponding columns/rows of the array $z$.

If $info = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

Each computed eigenvector $z_i$ is an exact eigenvector of a matrix $T+E_i$, where $||E_i||_2 = O(\epsilon) * ||T||_2$. However, a set of eigenvectors computed by this routine may not be orthogonal to so high a degree of accuracy as those computed by ?steqr.

**?disna**

*Computes the reciprocal condition numbers for the eigenvectors of a symmetric/Hermitian matrix or for the left or right singular vectors of a general matrix.*

**Syntax**

```c
lapack_int LAPACKE_sdisna (char job, lapack_int m, lapack_int n, const float* d, float* sep);

lapack_int LAPACKE_ddisna (char job, lapack_int m, lapack_int n, const double* d, double* sep);
```

**Include Files**

- mkl.h

**Description**

The routine computes the reciprocal condition numbers for the eigenvectors of a real symmetric or complex Hermitian matrix or for the left or right singular vectors of a general $m$-by-$n$ matrix.

The reciprocal condition number is the 'gap' between the corresponding eigenvalue or singular value and the nearest other one.

The bound on the error, measured by angle in radians, in the $i$-th computed vector is given by

$$\text{?lamch('E')} \times \frac{\text{anorm}}{\text{sep}(i)}$$

where $\text{anorm} = ||A||_2 = \max(|d(j)|)$. $\text{sep}(i)$ is not allowed to be smaller than $\text{slamch('E')} \times \text{anorm}$ in order to limit the size of the error bound.

?disna may also be used to compute error bounds for eigenvectors of the generalized symmetric definite eigenproblem.

**Input Parameters**

*job*

Must be 'E', 'L', or 'R'. Specifies for which problem the reciprocal condition numbers should be computed:

- $job = 'E'$: for the eigenvectors of a symmetric/Hermitian matrix;
- $job = 'L'$: for the left singular vectors of a general matrix;
- $job = 'R'$: for the right singular vectors of a general matrix.

*m*

The number of rows of the matrix ($m \geq 0$).

*n*

If $job = 'L'$, or 'R', the number of columns of the matrix ($n \geq 0$). Ignored if $job = 'E$.'
Array, dimension at least \( \max(1,m) \) if \( \text{job} = 'E' \), and at least \( \max(1,\min(m,n)) \) if \( \text{job} = 'L' \) or \( 'R' \).

This array must contain the eigenvalues (if \( \text{job} = 'E' \)) or singular values (if \( \text{job} = 'L' \) or \( 'R' \)) of the matrix, in either increasing or decreasing order.

If singular values, they must be non-negative.

Output Parameters

Array, dimension at least \( \max(1,m) \) if \( \text{job} = 'E' \), and at least \( \max(1,\min(m,n)) \) if \( \text{job} = 'L' \) or \( 'R' \). The reciprocal condition numbers of the vectors.

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Generalized Symmetric-Definite Eigenvalue Problems: LAPACK Computational Routines

Generalized symmetric-definite eigenvalue problems are as follows: find the eigenvalues \( \lambda \) and the corresponding eigenvectors \( z \) that satisfy one of these equations:

\[
Az = \lambda Bz, \quad ABz = \lambda z, \quad BAZ = \lambda z,
\]

where \( A \) is an \( n \)-by-\( n \) symmetric or Hermitian matrix, and \( B \) is an \( n \)-by-\( n \) symmetric positive-definite or Hermitian positive-definite matrix.

In these problems, there exist \( n \) real eigenvectors corresponding to real eigenvalues (even for complex Hermitian matrices \( A \) and \( B \)).

Routines described in this section allow you to reduce the above generalized problems to standard symmetric eigenvalue problem \( Cy = \lambda y \), which you can solve by calling LAPACK routines described earlier in this chapter (see Symmetric Eigenvalue Problems).

Different routines allow the matrices to be stored either conventionally or in packed storage. Prior to reduction, the positive-definite matrix \( B \) must first be factorized using either \texttt{potrf} or \texttt{pptrf}.

The reduction routine for the banded matrices \( A \) and \( B \) uses a split Cholesky factorization for which a specific routine \texttt{pbstf} is provided. This refinement halves the amount of work required to form matrix \( C \).

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists LAPACK routines that can be used to solve generalized symmetric-definite eigenvalue problems.

### Computational Routines for Reducing Generalized Eigenproblems to Standard Problems

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>Reduce to standard problems (full storage)</th>
<th>Reduce to standard problems (packed storage)</th>
<th>Reduce to standard problems (band matrices)</th>
<th>Factorize band matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>real symmetric matrices</td>
<td>sygst</td>
<td>spgst</td>
<td>sbgst</td>
<td>pbstf</td>
</tr>
<tr>
<td>complex Hermitian matrices</td>
<td>hegst</td>
<td>hpgst</td>
<td>hbgst</td>
<td>pbstf</td>
</tr>
</tbody>
</table>
Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

**Syntax**

```c
lapack_int LAPACKE_ssygst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, float* a, lapack_int lda, const float* b, lapack_int ldb);

lapack_int LAPACKE_dsygst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, double* a, lapack_int lda, const double* b, lapack_int ldb);
```

**Include Files**

- `mkl.h`

**Description**

The routine reduces real symmetric-definite generalized eigenproblems

\[ A z = \lambda B z, \ A^T B z = \lambda z, \text{ or } B A^T z = \lambda z \]

to the standard form \( Cy = \lambda y \). Here \( A \) is a real symmetric matrix, and \( B \) is a real symmetric positive-definite matrix. Before calling this routine, call `?potrf` to compute the Cholesky factorization: \( B = U^T U \) or \( B = L L^T \).

**Input Parameters**

- `matrix_layout`
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `itype`
  
  Must be 1 or 2 or 3.

  - **If itype = 1**, the generalized eigenproblem is \( A z = \lambda B z \)
    
    for `uplo = 'U'`: \( C = \text{inv}(U^T) A \text{inv}(U), z = \text{inv}(U) y \);
    
    for `uplo = 'L'`: \( C = \text{inv}(L) A \text{inv}(L^T), z = \text{inv}(L^T) y \).

  - **If itype = 2**, the generalized eigenproblem is \( A^T B z = \lambda z \)
    
    for `uplo = 'U'`: \( C = U^T A U^T, z = \text{inv}(U) y \);
    
    for `uplo = 'L'`: \( C = L^T A L, z = \text{inv}(L^T) y \).

  - **If itype = 3**, the generalized eigenproblem is \( B A^T z = \lambda z \)
    
    for `uplo = 'U'`: \( C = U^T A^T U, z = U^T y \);
    
    for `uplo = 'L'`: \( C = L^T A^T L, z = L^T y \).

- `uplo`
  
  Must be 'U' or 'L'.

  - If `uplo = 'U'`, the array \( a \) stores the upper triangle of \( A \); you must supply \( B \) in the factored form \( B = U^T U \).
  
  - If `uplo = 'L'`, the array \( a \) stores the lower triangle of \( A \); you must supply \( B \) in the factored form \( B = L^T L \).

- `n`
  
  The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

- `a, b`
  
  Arrays:

  \( a \) (size \( \max(1, lda \times n) \)) contains the upper or lower triangle of \( A \).
$b$ (size max(1, $ldb \cdot n$)) contains the Cholesky-factored matrix $B$:

\[ B = U^T \cdot U \text{ or } B = L \cdot L^T \text{ (as returned by } \texttt{?potrf}). \]

$lda$
The leading dimension of $a$; at least max(1, \(n\)).

$ldb$
The leading dimension of $b$; at least max(1, \(n\)).

**Output Parameters**

$a$
The upper or lower triangle of $A$ is overwritten by the upper or lower triangle of $C$, as specified by the arguments $itype$ and $uplo$.

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

Forming the reduced matrix $C$ is a stable procedure. However, it involves implicit multiplication by $\text{inv}(B)$ (if $itype = 1$) or $B$ (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if $B$ is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is $n^3$.

**?hegst**

*Reduces a complex Hermitian positive-definite generalized eigenvalue problem to the standard form.*

**Syntax**

```c
lapack_int LAPACKE_chegst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* b, lapack_int ldb);
```

```c
lapack_int LAPACKE_zhegst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The routine reduces a complex Hermitian positive-definite generalized eigenvalue problem to standard form.

<table>
<thead>
<tr>
<th>$itype$</th>
<th>Problem</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A x = \lambda B x$</td>
<td>$A$ overwritten by $\text{inv}(U^H) \cdot A \cdot \text{inv}(U)$ or $\text{inv}(L) \cdot A \cdot \text{inv}(L^H)$</td>
</tr>
<tr>
<td>2</td>
<td>$A B x = \lambda x$</td>
<td>$A$ overwritten by $U^H \cdot A \cdot U$ or $L^H \cdot A \cdot L$</td>
</tr>
<tr>
<td>3</td>
<td>$B A x = \lambda x$</td>
<td></td>
</tr>
</tbody>
</table>
Before calling this routine, you must call ?potrf to compute the Cholesky factorization: \( B = U^H U \) or \( B = L^H L \).

### Input Parameters

**itype**

Must be 1 or 2 or 3.

- If \( itype = 1 \), the generalized eigenproblem is \( Ax = \lambda Bx \)
  - for \( uplo = 'U' \): \( C = (U^H)^{-1} A U^{-1} \)
  - for \( uplo = 'L' \): \( C = L^{-1} A (L^H)^{-1} \)

- If \( itype = 2 \), the generalized eigenproblem is \( ABx = \lambda x \)
  - for \( uplo = 'U' \): \( C = U^H A U \)
  - for \( uplo = 'L' \): \( C = L H A L \)

- If \( itype = 3 \), the generalized eigenproblem is \( Bx = \lambda x \)
  - for \( uplo = 'U' \): \( C = U^H A U \)
  - for \( uplo = 'L' \): \( C = L H A L \)

**uplo**

Must be 'U' or 'L'.

- If \( uplo = 'U' \), the array \( a \) stores the upper triangle of \( A \); you must supply \( B \) in the factored form \( B = U^H U \).
- If \( uplo = 'L' \), the array \( a \) stores the lower triangle of \( A \); you must supply \( B \) in the factored form \( B = L^H L \).

**n**

The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

**a, b**

Arrays:

- \( a \) (size \( \max(1, lda \times n) \)) contains the upper or lower triangle of \( A \).
- \( b \) (size \( \max(1, ldb \times n) \)) contains the Cholesky-factored matrix \( B \): \( B = U^H U \) or \( B = L^H L \) (as returned by \( ?potrf \)).

**lda**

The leading dimension of \( a \); at least \( \max(1, n) \).

**ldb**

The leading dimension of \( b \); at least \( \max(1, n) \).

### Output Parameters

**a**

The upper or lower triangle of \( A \) is overwritten by the upper or lower triangle of \( C \), as specified by the arguments \( itype \) and \( uplo \).

### Return Values

This function returns a value \( info \).

- If \( info = 0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.
Application Notes

Forming the reduced matrix $C$ is a stable procedure. However, it involves implicit multiplication by $B^{-1}$ (if $itype = 1$) or $B$ (if $itype = 2$ or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if $B$ is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is $n^3$.

?spgst

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form using packed storage.

Syntax

```c
lapack_int LAPACKE_sspgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, float* ap, const float* bp);
```

```c
lapack_int LAPACKE_dspgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, double* ap, const double* bp);
```

Include Files

- mkl.h

Description

The routine reduces real symmetric-definite generalized eigenproblems

$$A^*x = \lambda B^*x, A^*B^*x = \lambda^*x, \text{ or } B^*A^*x = \lambda^*x$$

To the standard form $C^*y = \lambda^*y$, using packed matrix storage. Here $A$ is a real symmetric matrix, and $B$ is a real symmetric positive-definite matrix. Before calling this routine, call ?pptrf to compute the Cholesky factorization: $B = U^*U$ or $B = L^*L^T$.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `itype` Must be 1 or 2 or 3.
  - If $itype = 1$, the generalized eigenproblem is $A^*z = \lambda B^*z$
    - for `uplo = 'U'`: $C = \text{inv}(U^T)A^*\text{inv}(U), z = \text{inv}(U)^*y$;
    - for `uplo = 'L'`: $C = \text{inv}(L)A^*\text{inv}(L^T), z = \text{inv}(L^T)^*y$.
  - If $itype = 2$, the generalized eigenproblem is $A^*B^*z = \lambda^*z$
    - for `uplo = 'U'`: $C = U^*A^*U^T, z = \text{inv}(U)^*y$;
    - for `uplo = 'L'`: $C = L^T*A^*L, z = \text{inv}(L^T)^*y$.
  - If $itype = 3$, the generalized eigenproblem is $B^*A^*z = \lambda^*z$
    - for `uplo = 'U'`: $C = U^*A^*U^T, z = U^T*y$;
    - for `uplo = 'L'`: $C = L^T*A^*L, z = L*y$.
- `uplo` Must be 'U' or 'L'.
If \( \text{uplo} = 'U' \), \( ap \) stores the packed upper triangle of \( A \); you must supply \( B \) in the factored form \( B = U^T \cdot U \).

If \( \text{uplo} = 'L' \), \( ap \) stores the packed lower triangle of \( A \); you must supply \( B \) in the factored form \( B = L \cdot L^T \).

\( n \)  
The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

\( ap, bp \)  
Arrays:
- \( ap \) contains the packed upper or lower triangle of \( A \).
- The dimension of \( ap \) must be at least \( \max(1, n \cdot (n+1)/2) \).
- \( bp \) contains the packed Cholesky factor of \( B \) (as returned by \(?\text{pptrf}\) with the same \( \text{uplo} \) value).
- The dimension of \( bp \) must be at least \( \max(1, n \cdot (n+1)/2) \).

**Output Parameters**

- \( ap \)  
The upper or lower triangle of \( A \) is overwritten by the upper or lower triangle of \( C \), as specified by the arguments \( \text{itype} \) and \( \text{uplo} \).

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

Forming the reduced matrix \( C \) is a stable procedure. However, it involves implicit multiplication by \( \text{inv}(B) \) (if \( \text{itype} = 1 \)) or \( B \) (if \( \text{itype} = 2 \) or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if \( B \) is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is \( n^3 \).

**\(?\text{hpgst}\)**  
Reduces a generalized eigenvalue problem with a Hermitian matrix to a standard eigenvalue problem using packed storage.

**Syntax**

```c
lapack_int LAPACKE_chpgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_float* ap, const lapack_complex_float* bp);

lapack_int LAPACKE_zhpgst (int matrix_layout, lapack_int itype, char uplo, lapack_int n, lapack_complex_double* ap, const lapack_complex_double* bp);
```

**Include Files**

- \( \text{mkl.h} \)

**Description**

The routine reduces generalized eigenproblems with Hermitian matrices.
\[ A^*z = \lambda B^*z, \quad A^*B^*z = \lambda^*z, \quad \text{or} \quad B^*A^*z = \lambda^*z. \]

to standard eigenproblems \( C^*y = \lambda^*y \), using packed matrix storage. Here \( A \) is a complex Hermitian matrix, and \( B \) is a complex Hermitian positive-definite matrix. Before calling this routine, you must call \?pptrf to compute the Cholesky factorization: \( B = U^H*U \) or \( B = L*L^H \).

**Input Parameters**

**matrix_layout**

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**itype**

Must be 1 or 2 or 3.

If \( itype = 1 \), the generalized eigenproblem is \( A^*z = \lambda B^*z \)
for \( uplo = 'U' \): \( C = \text{inv}(U^H)*A*\text{inv}(U), \quad z = \text{inv}(U)^*y \);
for \( uplo = 'L' \): \( C = \text{inv}(L^H)*A*\text{inv}(L), \quad z = \text{inv}(L^H)^*y \).

If \( itype = 2 \), the generalized eigenproblem is \( A^*B^*z = \lambda^*z \)
for \( uplo = 'U' \): \( C = U*U^H*A, \quad z = \text{inv}(U)^*y \);
for \( uplo = 'L' \): \( C = L*L^H*A, \quad z = \text{inv}(L^H)^*y \).

If \( itype = 3 \), the generalized eigenproblem is \( B^*A^*z = \lambda^*z \)
for \( uplo = 'U' \): \( C = U^H*A*U, \quad z = U^H*y \);
for \( uplo = 'L' \): \( C = L*L^H*A, \quad z = L*y \).

**uplo**

Must be \( 'U' \) or \( 'L' \).

If \( uplo = 'U' \), \( ap \) stores the packed upper triangle of \( A \); you must supply \( B \) in the factored form \( B = U^H*U \).

If \( uplo = 'L' \), \( ap \) stores the packed lower triangle of \( A \); you must supply \( B \) in the factored form \( B = L*L^H \).

**n**

The order of the matrices \( A \) and \( B \) \((n \geq 0)\).

**ap, bp**

Arrays:

\( ap \) contains the packed upper or lower triangle of \( A \).

The dimension of \( a \) must be at least \( \max(1, n*(n+1)/2) \).

\( bp \) contains the packed Cholesky factor of \( B \) (as returned by \?pptrf with the same \( uplo \) value).

The dimension of \( b \) must be at least \( \max(1, n*(n+1)/2) \).

**Output Parameters**

**ap**

The upper or lower triangle of \( A \) is overwritten by the upper or lower triangle of \( C \), as specified by the arguments \( itype \) and \( uplo \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.
Application Notes

Forming the reduced matrix \( C \) is a stable procedure. However, it involves implicit multiplication by \( \text{inv}(B) \) (if \( \text{itype} = 1 \)) or \( B \) (if \( \text{itype} = 2 \) or 3). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if \( B \) is ill-conditioned with respect to inversion.

The approximate number of floating-point operations is \( n^3 \).

?sbgst

Reduces a real symmetric-definite generalized eigenproblem for banded matrices to the standard form using the factorization performed by ?pbstf.

Syntax

```c
lapack_int LAPACKE_ssbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, const float* bb, lapack_int lddb, float* x, lapack_int ldx);
```

Include Files

- mkl.h

Description

To reduce the real symmetric-definite generalized eigenproblem \( A*z = \lambda*B*z \) to the standard form \( C*y = \lambda*y \), where \( A, B \) and \( C \) are banded, this routine must be preceded by a call to ?pbstf, which computes the split Cholesky factorization of the positive-definite matrix \( B: B = S^T*S \). The split Cholesky factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites \( A \) with \( C = X^T*A*X \), where \( X = \text{inv}(S)^T*Q \) and \( Q \) is an orthogonal matrix chosen (implicitly) to preserve the bandwidth of \( A \). The routine also has an option to allow the accumulation of \( X \), and then, if \( z \) is an eigenvector of \( C \), \( X^T*z \) is an eigenvector of the original system.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `vect` Must be 'N' or 'V'.
  - If `vect` = 'N', then matrix \( X \) is not returned;
  - If `vect` = 'V', then matrix \( X \) is returned.
- `uplo` Must be 'U' or 'L'.
  - If `uplo` = 'U', \( ab \) stores the upper triangular part of \( A \).
  - If `uplo` = 'L', \( ab \) stores the lower triangular part of \( A \).
- `n` The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).
- `ka` The number of super- or sub-diagonals in \( A \).
\( (ka \geq 0). \)

\( kb \)

The number of super- or sub-diagonals in \( B \)
\( (ka \geq kb \geq 0). \)

\( ab, bb \)

\( ab \) (size at least \( \max(1, ldab*n) \) for column major layout and at least \( \max(1, ldab*(ka + 1)) \) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix \( A \) (as specified by \( uplo \)) in band storage format.

\( bb \) (size at least \( \max(1, ldbb*n) \) for column major layout and at least \( \max(1, ldbb*(kb + 1)) \) for row major layout) is an array containing the banded split Cholesky factor of \( B \) as specified by \( uplo \), \( n \) and \( kb \) and returned by \texttt{pbstf/pbstf}. 

\( ldab \)

The leading dimension of the array \( ab \); must be at least \( ka+1 \) for column major layout and \( \max(1, n) \) for row major layout.

\( ldbb \)

The leading dimension of the array \( bb \); must be at least \( kb+1 \) for column major layout and \( \max(1, n) \) for row major layout.

\( ldx \)

The leading dimension of the output array \( x \). Constraints: if \( vect = 'N' \), then \( ldx \geq 1; \) if \( vect = 'V', then ldx \geq \max(1, n). \)

Output Parameters

\( ab \)

On exit, this array is overwritten by the upper or lower triangle of \( C \) as specified by \( uplo \).

\( x \)

Array.

If \( vect = 'V', then x \) (size at least \( \max(1, ldx*n) \)) contains the \( n \)-by-\( n \) matrix \( X = \text{inv}(S)*Q. \)

If \( vect = 'N', then x \) is not referenced.

Return Values

This function returns a value \( info. \)

If \( info=0, the execution is successful. \)

If \( info = -i, the i-th parameter had an illegal value. \)

Application Notes

Forming the reduced matrix \( C \) involves implicit multiplication by \( \text{inv}(B) \). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if \( B \) is ill-conditioned with respect to inversion.

If \( ka \) and \( kb \) are much less than \( n \) then the total number of floating-point operations is approximately \( 6n^2*kb, \) when \( vect = 'N'. Additional \( (3/2)n^3*(kb/ka) \) operations are required when \( vect = 'V'. \)
?hbgst
Reduces a complex Hermitian positive-definite
generalized eigenproblem for banded matrices to the
standard form using the factorization performed
by ?pbstf.

Syntax

lapack_int LAPACKE_chbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab, const
lapack_complex_float* bb, lapack_int ldbb, lapack_complex_float* x, lapack_int ldx);

lapack_int LAPACKE_zhbgst (int matrix_layout, char vect, char uplo, lapack_int n,
lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab, const
lapack_complex_double* bb, lapack_int ldbb, lapack_complex_double* x, lapack_int ldx);

Include Files

• mkl.h

Description

To reduce the complex Hermitian positive-definite generalized eigenproblem \( A*z = \lambda*B*z \) to the standard
form \( C*x = \lambda*y \), where \( A, B \) and \( C \) are banded, this routine must be preceded by a call to pbstf/pbstf, which
computes the split Cholesky factorization of the positive-definite matrix \( B = S^H*S \). The split Cholesky
factorization, compared with the ordinary Cholesky factorization, allows the work to be approximately halved.

This routine overwrites \( A \) with \( C = X^H*A*X \), where \( X = \text{inv}(S)*Q \), and \( Q \) is a unitary matrix chosen
(implicitly) to preserve the bandwidth of \( A \). The routine also has an option to allow the accumulation of \( X \),
and then, if \( z \) is an eigenvector of \( C \), \( X*z \) is an eigenvector of the original system.

Input Parameters

matrix_layout          Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
                        or column major (LAPACK_COL_MAJOR).

vect                   Must be 'N' or 'V'.
                        If vect = 'N', then matrix \( X \) is not returned;
                        If vect = 'V', then matrix \( X \) is returned.

uplo                   Must be 'U' or 'L'.
                        If uplo = 'U', \( ab \) stores the upper triangular part of \( A \).
                        If uplo = 'L', \( ab \) stores the lower triangular part of \( A \).

n                       The order of the matrices \( A \) and \( B \) \((n \geq 0)\).

ka                      The number of super- or sub-diagonals in \( A \)
                        \((ka \geq 0)\).

kb                      The number of super- or sub-diagonals in \( B \)
                        \((ka \geq kb \geq 0)\).

ab, bb                  \( ab \)(size at least max(1, ldab*n) for column major layout and at least
                        max(1, ldab*(ka + 1)) for row major layout) is an array containing either
                        upper or lower triangular part of the Hermitian matrix \( A \) (as specified by
                        uplo) in band storage format.
bb (size at least max(1, lddb * n) for column major layout and at least \( \max(1, ldbb \cdot (kb + 1)) \) for row major layout) is an array containing the banded split Cholesky factor of \( B \) as specified by uplo, n and kb and returned by pbstf/pbstf.

\( ldab \)

The leading dimension of the array \( ab \); must be at least \( ka + 1 \) for column major layout and \( \max(1, n) \) for row major layout.

\( ldbb \)

The leading dimension of the array \( bb \); must be at least \( kb + 1 \) for column major layout and \( \max(1, n) \) for row major layout.

\( ldx \)

The leading dimension of the output array \( x \). Constraints:

- If \( vect = 'N' \), then \( ldx \geq 1 \);
- If \( vect = 'V' \), then \( ldx \geq \max(1, n) \).

**Output Parameters**

\( ab \)

On exit, this array is overwritten by the upper or lower triangle of \( C \) as specified by uplo.

\( x \)

Array.

- If \( vect = 'V' \), then \( x \) (size at least \( \max(1, ldx \cdot n) \)) contains the \( n \)-by-\( n \) matrix \( X = \text{inv}(S) \cdot Q \).
- If \( vect = 'N' \), then \( x \) is not referenced.

**Return Values**

This function returns a value \( info \).

- If \( info = 0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

Forming the reduced matrix \( C \) involves implicit multiplication by \( \text{inv}(B) \). When the routine is used as a step in the computation of eigenvalues and eigenvectors of the original problem, there may be a significant loss of accuracy if \( B \) is ill-conditioned with respect to inversion. The total number of floating-point operations is approximately \( 20n^2 \cdot kb \), when \( vect = 'N' \). Additional \( 5n^3 \cdot (kb/ka) \) operations are required when \( vect = 'V' \). All these estimates assume that both \( ka \) and \( kb \) are much less than \( n \).

**pbstf**

*Computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-definite banded matrix used in *?sbgst/?hbgst.*

**Syntax**

```c
lapack_int LAPACKE_spbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb, float* bb, lapack_int ldbb);
lapack_int LAPACKE_dpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb, double* bb, lapack_int ldbb);
lapack_int LAPACKE_cpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb, lapack_complex_float* bb, lapack_int ldbb);
```
lapack_int LAPACKE_zpbstf (int matrix_layout, char uplo, lapack_int n, lapack_int kb, lapack_complex_double* bb, lapack_int ldbb);

Include Files
• mkl.h

Description
The routine computes a split Cholesky factorization of a real symmetric or complex Hermitian positive-
definite band matrix $B$. It is to be used in conjunction with sbgst/hbgst.

The factorization has the form $B = S^T S$ (or $B = S^H S$ for complex flavors), where $S$ is a band matrix of the same bandwidth as $B$ and the following structure: $S$ is upper triangular in the first $(n+kb)/2$ rows and lower triangular in the remaining rows.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

uplo
Must be ’U’ or ’L’.
If $uplo = ’U’$, $bb$ stores the upper triangular part of $B$.
If $uplo = ’L’$, $bb$ stores the lower triangular part of $B$.

n
The order of the matrix $B$ ($n \geq 0$).

kb
The number of super- or sub-diagonals in $B$ ($kb \geq 0$).

bb
$bb$ (size at least max(1, ldbb*n) for column major layout and at least max(1, ldbb*(kb + 1)) for row major layout) is an array containing either upper or lower triangular part of the matrix $B$ (as specified by $uplo$) in band storage format.

ldbb
The leading dimension of $bb$; must be at least $kb+1$ for column major and at least max(1, $n$) for row major.

Output Parameters

bb
On exit, this array is overwritten by the elements of the split Cholesky factor $S$.

Return Values
This function returns a value $info$.
If $info=0$, the execution is successful.
If $info = i$, then the factorization could not be completed, because the updated element $b_{ii}$ would be the square root of a negative number; hence the matrix $B$ is not positive-definite.
If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes
The computed factor $S$ is the exact factor of a perturbed matrix $B + E$, where
\[ |E| \leq c(kb + 1)\varepsilon|S^H||S|, \quad |e_{ij}| \leq c(kb + 1)\varepsilon\sqrt{b_{ii}b_{jj}} \]

c(n) is a modest linear function of \( n \), and \( \varepsilon \) is the machine precision.

The total number of floating-point operations for real flavors is approximately \( n(kb+1)^2 \). The number of operations for complex flavors is 4 times greater. All these estimates assume that \( kb \) is much less than \( n \).

After calling this routine, you can call sbgst/hbgst to solve the generalized eigenproblem \( Ax = \lambda Bz \), where \( A \) and \( B \) are banded and \( B \) is positive-definite.

**Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines**

This section describes LAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

A nonsymmetric eigenvalue problem is as follows: given a nonsymmetric (or non-Hermitian) matrix \( A \), find the eigenvalues \( \lambda \) and the corresponding eigenvectors \( z \) that satisfy the equation

\[ Az = \lambda z \]  
(right eigenvectors \( z \))

or the equation

\[ z^H A = \lambda z^H \]  
(left eigenvectors \( z \)).

Nonsymmetric eigenvalue problems have the following properties:

- The number of eigenvectors may be less than the matrix order (but is not less than the number of distinct eigenvalues of \( A \)).
- Eigenvalues may be complex even for a real matrix \( A \).
- If a real nonsymmetric matrix has a complex eigenvalue \( a+bi \) corresponding to an eigenvector \( z \), then \( a-bi \) is also an eigenvalue. The eigenvalue \( a-bi \) corresponds to the eigenvector whose elements are complex conjugate to the elements of \( z \).

To solve a nonsymmetric eigenvalue problem with LAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained. Table "Computational Routines for Solving Nonsymmetric Eigenvalue Problems" lists LAPACK routines to reduce the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation \( A = QH^H \) as well as routines to solve eigenvalue problems with Hessenberg matrices, forming the Schur factorization of such matrices and computing the corresponding condition numbers.

The decision tree in Figure "Decision Tree: Real Nonsymmetric Eigenvalue Problems" helps you choose the right routine or sequence of routines for an eigenvalue problem with a real nonsymmetric matrix. If you need to solve an eigenvalue problem with a complex non-Hermitian matrix, use the decision tree shown in Figure "Decision Tree: Complex Non-Hermitian Eigenvalue Problems".

**Computational Routines for Solving Nonsymmetric Eigenvalue Problems**

<table>
<thead>
<tr>
<th>Operation performed</th>
<th>Routines for real matrices</th>
<th>Routines for complex matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce to Hessenberg form ( A = QH^H )</td>
<td>?gehrd,</td>
<td>?gehrd</td>
</tr>
<tr>
<td>Generate the matrix ( Q )</td>
<td>?orgqr</td>
<td>?unghr</td>
</tr>
<tr>
<td>Apply the matrix ( Q )</td>
<td>?ormqr</td>
<td>?unmqr</td>
</tr>
<tr>
<td>Balance matrix</td>
<td>?gebal</td>
<td>?gebal</td>
</tr>
<tr>
<td>Transform eigenvectors of balanced matrix to those of the original matrix</td>
<td>?gebak</td>
<td>?gebak</td>
</tr>
<tr>
<td>Find eigenvalues and Schur factorization (QR algorithm)</td>
<td>?hseqr</td>
<td>?hseqr</td>
</tr>
<tr>
<td>Operation performed</td>
<td>Routines for real matrices</td>
<td>Routines for complex matrices</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Find eigenvectors from Hessenberg form (inverse iteration)</td>
<td>?hsein</td>
<td>?hsein</td>
</tr>
<tr>
<td>Find eigenvectors from Schur factorization</td>
<td>?trevc</td>
<td>?trevc</td>
</tr>
<tr>
<td>Estimate sensitivities of eigenvalues and eigenvectors</td>
<td>?trsna</td>
<td>?trsna</td>
</tr>
<tr>
<td>Reorder Schur factorization</td>
<td>?trexc</td>
<td>?trexc</td>
</tr>
<tr>
<td>Reorder Schur factorization, find the invariant subspace and estimate sensitivities</td>
<td>?trsen</td>
<td>?trsen</td>
</tr>
<tr>
<td>Solves Sylvester’s equation.</td>
<td>?trsyl</td>
<td>?trsyl</td>
</tr>
</tbody>
</table>

**Decision Tree: Real Nonsymmetric Eigenvalue Problems**

1. Are eigenvalues only required?  
   - yes  
     - Is A an upper Hessenberg matrix?  
       - yes  
         - ?HSEQR  
       - no  
         - ?GEBAL ?GEHRD ?HSEQR
   - no  
     - ?GEBAL ?GEHRD

2. Is the Schur factorization of A required?  
   - yes  
     - Is A an upper Hessenberg matrix?  
       - yes  
         - ?HSEQR  
       - no  
   - no  

3. Are all eigenvectors required?  
   - yes  
     - Is A an upper Hessenberg matrix?  
       - yes  
         - ?HSEQR ?TREVC  
       - no  
   - no  

4. Is A an upper Hessenberg matrix?  
   - yes  
     - ?HSEQR ?HSEIN  
   - no  
**Decision Tree: Complex Non-Hermitian Eigenvalue Problems**

- **Are eigenvalues only required?**
  - **yes**
    - **Is A an upper Hessenberg matrix?**
      - **yes**
        - ?HSEQR
      - **no**
        - ?GEBAL ?GEHRD ?HSEQR
  - **no**
    - ?GEBAL ?GEHRD

- **Is the Schur factorization of A required?**
  - **yes**
    - **Is A an upper Hessenberg matrix?**
      - **yes**
        - ?HSEQR
      - **no**
  - **no**
    - ?GEBAL ?GEHRD

- **Are all eigenvectors required?**
  - **yes**
    - **Is A an upper Hessenberg matrix?**
      - **yes**
        - ?HSEQR ?TREVC
      - **no**
  - **no**
    - ?GEBAL ?GEHRD

?gehrd

*Reduces a general matrix to upper Hessenberg form.*

**Syntax**

```c
lapack_int LAPACKE_sgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, float* a, lapack_int lda, float* tau);

lapack_int LAPACKE_dgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, double* a, lapack_int lda, double* tau);

lapack_int LAPACKE_cgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgehrd (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```
Include Files
• mkl.h

Description
The routine reduces a general matrix $A$ to upper Hessenberg form $H$ by an orthogonal or unitary similarity transformation $A = Q^*HQ$. Here $H$ has real subdiagonal elements.

The routine does not form the matrix $Q$ explicitly. Instead, $Q$ is represented as a product of elementary reflectors. Routines are provided to work with $Q$ in this representation.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

$n$
The order of the matrix $A (n \geq 0)$.

$ilo, ihi$
If $A$ is an output by ?gebal, then $ilo$ and $ihi$ must contain the values returned by that routine. Otherwise $ilo = 1$ and $ihi = n$. (If $n > 0$, then $1 \leq ilo \leq ihi \leq n$; if $n = 0$, $ilo = 1$ and $ihi = 0$.)

$a$
Arrays: $a$ (size max(1, lda*n)) contains the matrix $A$.

lda
The leading dimension of $a$; at least max(1, $n$).

Output Parameters

$a$
The elements on and above the subdiagonal contain the upper Hessenberg matrix $H$. The subdiagonal elements of $H$ are real. The elements below the subdiagonal, with the array $tau$, represent the orthogonal matrix $Q$ as a product of $n$ elementary reflectors.

$tau$
Array, size at least max (1, $n$-1).

Contains scalars that define elementary reflectors for the matrix $Q$.

Return Values
This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes
The computed Hessenberg matrix $H$ is exactly similar to a nearby matrix $A + E$, where $||E||_2 < c(n)\varepsilon||A||_2$, $c(n)$ is a modestly increasing function of $n$, and $\varepsilon$ is the machine precision.

The approximate number of floating-point operations for real flavors is $(2/3)*(ihi - ilo)^2(2ihi + 2ilo + 3n)$; for complex flavors it is 4 times greater.

?orghr
Generates the real orthogonal matrix $Q$ determined by ?gehrd.
Syntax

```c
lapack_int LAPACKE_sorghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, float* a, lapack_int lda, const float* tau);

lapack_int LAPACKE_dorghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, double* a, lapack_int lda, const double* tau);
```

Include Files

- mkl.h

Description

The routine explicitly generates the orthogonal matrix \( Q \) that has been determined by a preceding call to \( \text{sgehrd} / \text{dgehrd} \). (The routine \( \text{sgehrd} \) reduces a real general matrix \( A \) to upper Hessenberg form \( H \) by an orthogonal similarity transformation, \( A = Q^*H^*Q \), and represents the matrix \( Q \) as a product of \( ihi-ilo \) elementary reflectors. Here \( ilo \) and \( ihi \) are values determined by \( \text{sgebal} / \text{dgebal} \) when balancing the matrix; if the matrix has not been balanced, \( ilo = 1 \) and \( ihi = n \).)

The matrix \( Q \) generated by \( \text{orghr} \) has the structure:

\[
Q = \begin{bmatrix}
I & 0 & 0 \\
0 & Q_{22} & 0 \\
0 & 0 & I
\end{bmatrix}
\]

where \( Q_{22} \) occupies rows and columns \( ilo \) to \( ihi \).

Input Parameters

- \textit{matrix_layout} Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \textit{n} The order of the matrix \( Q (n \geq 0) \).
- \textit{ilo, ihi} These must be the same parameters \( ilo \) and \( ihi \), respectively, as supplied to \( \text{sgehrd} \). (If \( n > 0 \), then \( 1 \leq ilo \leq ihi \leq n \); if \( n = 0 \), \( ilo = 1 \) and \( ihi = 0 \).)
- \textit{a, tau} Arrays: \( a \) (size max(1, \( lda \times n \))) contains details of the vectors which define the elementary reflectors, as returned by \( \text{sgehrd} \).
- \( tau \) contains further details of the elementary reflectors, as returned by \( \text{sgehrd} \).
- \textit{lda} The dimension of \( tau \) must be at least max \( (1, n-1) \).

Output Parameters

- \textit{a} Overwritten by the \( n \)-by-\( n \) orthogonal matrix \( Q \).

Return Values

This function returns a value \( info \).
If $\text{info}=0$, the execution is successful.
If $\text{info} = -i$, the $i$-th parameter had an illegal value.

Application Notes
The computed matrix $Q$ differs from the exact result by a matrix $E$ such that $||E||_2 = O(\epsilon)$, where $\epsilon$ is the machine precision.
The approximate number of floating-point operations is $(4/3)(ihi-ilo)^3$.
The complex counterpart of this routine is unghr.

**?ormhr**
*Multiplies an arbitrary real matrix C by the real orthogonal matrix Q determined by ?gehrd.*

**Syntax**
```c
lapack_int LAPACKE_sormhr (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int ilo, lapack_int ihi, const float* a, lapack_int lda, const float* tau, float* c, lapack_int ldc);
lapack_int LAPACKE_dormhr (int matrix_layout, char side, char trans, lapack_int m, lapack_int n, lapack_int ilo, lapack_int ihi, const double* a, lapack_int lda, const double* tau, double* c, lapack_int ldc);
```

**Include Files**
- mkl.h

**Description**
The routine multiplies a matrix $C$ by the orthogonal matrix $Q$ that has been determined by a preceding call to sgehrd/dgehrd. (The routine ?gehrd reduces a real general matrix $A$ to upper Hessenberg form $H$ by an orthogonal similarity transformation, $A = Q^*H*Q^T$, and represents the matrix $Q$ as a product of $ihi$-$ilo$ elementary reflectors. Here $ilo$ and $ihi$ are values determined by sgebal/dgebal when balancing the matrix; if the matrix has not been balanced, $ilo = 1$ and $ihi = n$.)

With ?ormhr, you can form one of the matrix products $Q^*C$, $Q^T*C$, $C*Q$, or $C*Q^T$, overwriting the result on $C$ (which may be any real rectangular matrix).

A common application of ?ormhr is to transform a matrix $V$ of eigenvectors of $H$ to the matrix $QV$ of eigenvectors of $A$.

**Input Parameters**
- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **side**
  Must be 'L' or 'R'.
  If side= 'L', then the routine forms $Q^*C$ or $Q^T*C$.
  If side= 'R', then the routine forms $C*Q$ or $C*Q^T$.
- **trans**
  Must be 'N' or 'T'.
  If trans= 'N', then $Q$ is applied to $C$.
  If trans= 'T', then $Q^T$ is applied to $C$. 
The number of rows in $C$ ($m \geq 0$).

The number of columns in $C$ ($n \geq 0$).

These must be the same parameters $ilo$ and $ihi$, respectively, as supplied to `?gehrd`.

- If $m > 0$ and $side = 'L'$, then $1 \leq ilo \leq ihi \leq m$.
- If $m = 0$ and $side = 'L'$, then $ilo = 1$ and $ihi = 0$.
- If $n > 0$ and $side = 'R'$, then $1 \leq ilo \leq ihi \leq n$.
- If $n = 0$ and $side = 'R'$, then $ilo = 1$ and $ihi = 0$.

Arrays:

- $a$ (size max(1, $lda*n$) for $side='R$ and size max(1, $lda*m$) for $side='L$) contains details of the vectors which define the elementary reflectors, as returned by `?gehrd`.
- $tau$ contains further details of the elementary reflectors, as returned by `?gehrd`.

The dimension of $tau$ must be at least max (1, $m-1$) if $side = 'L'$ and at least max (1, $n-1$) if $side = 'R'$.

- $c$ (size max(1, $ldc*n$) for column major layout and max(1, $ldc*m$ for row major layout) contains the $m$ by $n$ matrix $C$.

The leading dimension of $a$; at least max(1, $m$) if $side = 'L'$ and at least max (1, $n$) if $side = 'R'$.

The leading dimension of $c$; at least max(1, $m$) for column major layout and at least max(1, $n$) for row major layout.

Output Parameters

$C$ is overwritten by product $Q^T C$, $Q^T C$, $C^T Q$, or $C^T Q^T$ as specified by $side$ and $trans$.

Return Values

This function returns a value $info$.

- If $info=0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.

Application Notes

The computed matrix $Q$ differs from the exact result by a matrix $E$ such that $||E||_2 = O(\varepsilon)||C||_2$, where $\varepsilon$ is the machine precision.

The approximate number of floating-point operations is

- $2n(ihi-ilo)^2$ if $side = 'L'$;
- $2m(ihi-ilo)^2$ if $side = 'R'$.

The complex counterpart of this routine is `unmhr`.

```
```
?unghr
Generates the complex unitary matrix Q determined by ?gehrd.

Syntax

lapack_int LAPACKE_cunghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* a, lapack_int lda, const lapack_complex_float* tau);

lapack_int LAPACKE_zunghr (int matrix_layout, lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* a, lapack_int lda, const lapack_complex_double* tau);

Include Files

• mkl.h

Description

The routine is intended to be used following a call to cgehrd/zgehrd, which reduces a complex matrix A to upper Hessenberg form H by a unitary similarity transformation: A = Q*H*Q^H. ?gehrd represents the matrix Q as a product of ihi-iloelementary reflectors. Here ilo and ihi are values determined by cgebal/zgebal when balancing the matrix; if the matrix has not been balanced, ilo = 1 and ihi = n.

Use the routine unghr to generate Q explicitly as a square matrix. The matrix Q has the structure:

\[
Q = \begin{bmatrix}
I & 0 & 0 \\
0 & Q_{22} & 0 \\
0 & 0 & I
\end{bmatrix}
\]

where Q_{22} occupies rows and columns ilo to ihi.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

n

The order of the matrix Q (n ≥ 0).

ilo, ihi

These must be the same parameters ilo and ihi, respectively, as supplied to ?gehrd. (If n > 0, then 1 ≤ ilo ≤ ihi ≤ n. If n = 0, then ilo = 1 and ihi = 0.)

a, tau

Arrays:

a (size max(1, lda*n)) contains details of the vectors which define the elementary reflectors, as returned by ?gehrd.

tau contains further details of the elementary reflectors, as returned by ?gehrd.

The dimension of tau must be at least max (1, n-1).

lda

The leading dimension of a; at least max(1, n).

Output Parameters

a

Overwritten by the n-by-n unitary matrix Q.
Return Values
This function returns a value \textit{info}.

If \textit{info}=0, the execution is successful.

If \textit{info} = -i, the i-th parameter had an illegal value.

Application Notes
The computed matrix \textit{Q} differs from the exact result by a matrix \textit{E} such that \|E\|_2 = O(\varepsilon), where \varepsilon is the machine precision.

The approximate number of real floating-point operations is \((16/3)(ihi-ilo)^3\).

The real counterpart of this routine is \texttt{orghr}.

\texttt{?unmhr}

\textit{Multiplies an arbitrary complex matrix }\textit{C} \textit{by the complex unitary matrix }\textit{Q} \textit{determined by } \texttt{?gehrd}.

Syntax
\begin{verbatim}
\end{verbatim}
\begin{verbatim}
lapack_int LAPACK_zunmhr (int \textit{matrix_layout}, char \textit{side}, char \textit{trans}, lapack_int \textit{m}, lapack_int \textit{n}, lapack_int \textit{ilo}, lapack_int \textit{ihi}, const lapack_complex_double* \textit{a}, lapack_int \textit{lda}, const lapack_complex_double* \textit{tau}, lapack_complex_double* \textit{c}, lapack_int \textit{ldc});
\end{verbatim}

Include Files
\begin{itemize}
  \item mkl.h
\end{itemize}

Description
The routine multiplies a matrix \textit{C} by the unitary matrix \textit{Q} that has been determined by a preceding call to \texttt{cgehrd/zgehrd}. (The routine \texttt{?gehrd} reduces a real general matrix \textit{A} to upper Hessenberg form \textit{H} by an orthogonal similarity transformation, \(A = Q^*HQ\), and represents the matrix \textit{Q} as a product of \textit{ihi-ilo} elementary reflectors. Here \textit{ilo} and \textit{ihi} are values determined by \texttt{cgebal/zgebal} when balancing the matrix; if the matrix has not been balanced, \textit{ilo} = 1 and \textit{ihi} = \textit{n}.)

With \texttt{?unmhr}, you can form one of the matrix products \(Q^*C\), \(Q^H*C\), \(C*Q\), or \(C*Q^H\), overwriting the result on \textit{C} (which may be any complex rectangular matrix). A common application of this routine is to transform a matrix \textit{V} of eigenvectors of \textit{H} to the matrix \textit{QV} of eigenvectors of \textit{A}.

Input Parameters
\begin{itemize}
  \item \textit{matrix_layout} \hspace{2cm} Specifies whether matrix storage layout is row major (\texttt{LAPACK_ROW_MAJOR}) or column major (\texttt{LAPACK_COL_MAJOR}).
  \item \textit{side} \hspace{2cm} Must be '\texttt{L}' or '\texttt{R}'.
    \hspace{1cm} If \textit{side} = '\texttt{L}', then the routine forms \(Q*C\) or \(Q^H*C\).
    \hspace{1cm} If \textit{side} = '\texttt{R}', then the routine forms \(C*Q\) or \(C*Q^H\).
  \item \textit{trans} \hspace{2cm} Must be '\texttt{N}' or '\texttt{C}'.
\end{itemize}
If \( \text{trans} = 'N' \), then \( Q \) is applied to \( C \).

If \( \text{trans} = 'T' \), then \( Q^H \) is applied to \( C \).

\( m \)

The number of rows in \( C \) (\( m \geq 0 \)).

\( n \)

The number of columns in \( C \) (\( n \geq 0 \)).

\( ilo, ihi \)

These must be the same parameters \( ilo \) and \( ihi \), respectively, as supplied to \(?\text{gehrd}\).

If \( m > 0 \) and \( \text{side} = 'L' \), then \( 1 \leq ilo \leq ihi \leq m \).

If \( m = 0 \) and \( \text{side} = 'L' \), then \( ilo = 1 \) and \( ihi = 0 \).

If \( n > 0 \) and \( \text{side} = 'L' \), then \( 1 \leq ilo \leq ihi \leq n \).

If \( n = 0 \) and \( \text{side} = 'R' \), then \( ilo =1 \) and \( ihi = 0 \).

\( a, \tau, c \)

Arrays:

\( a(\text{size max}(1,lda*n) \text{ for } \text{side} = 'R' \text{ and size max}(1,lda*m) \text{ for } \text{side} = 'L') \)

contains details of the vectors which define the elementary reflectors, as returned by \(?\text{gehrd}\).

\( \tau \) contains further details of the elementary reflectors, as returned by \(?\text{gehrd}\).

The dimension of \( \tau \) must be at least \( \text{max}(1, m-1) \)

if \( \text{side} = 'L' \) and at least \( \text{max}(1, n-1) \) if \( \text{side} = 'R' \).

\( c(\text{size max}(1, ldc*n) \text{ for column major layout and max}(1, ldc*m \text{ for row major layout}) \text{ contains the } m\text{-by-}n \text{ matrix } C. \)

\( lda \)

The leading dimension of \( a \); at least \( \text{max}(1, m) \) if \( \text{side} = 'L' \) and at least \( \text{max}(1, n) \) if \( \text{side} = 'R' \).

\( ldc \)

The leading dimension of \( c \); at least \( \text{max}(1, m) \) for column major layout and at least \( \text{max}(1, n) \) for row major layout.

**Output Parameters**

\( c \)

\( C \) is overwritten by \( Q*C \), or \( Q^H*C \), or \( C*Q^H \), or \( C*Q \) as specified by \( \text{side} \) and \( \text{trans} \).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The computed matrix \( Q \) differs from the exact result by a matrix \( E \) such that \( \|E\|_2 = O(\varepsilon) \|C\|_2 \), where \( \varepsilon \) is the machine precision.

The approximate number of floating-point operations is

\( 8n(\text{hi}-\text{ilo})^2 \) if \( \text{side} = 'L' \);

\( 8m(\text{hi}-\text{ilo})^2 \) if \( \text{side} = 'R' \).
The real counterpart of this routine is ormhr.

?gebal
Balances a general matrix to improve the accuracy of computed eigenvalues and eigenvectors.

Syntax

```c
lapack_int LAPACKE_sgebal( int matrix_layout, char job, lapack_int n, float* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, float* scale );
lapack_int LAPACKE_dgebal( int matrix_layout, char job, lapack_int n, double* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, double* scale );
lapack_int LAPACKE_cgebal( int matrix_layout, char job, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, float* scale );
lapack_int LAPACKE_zgebal( int matrix_layout, char job, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* ilo, lapack_int* ihi, double* scale );
```

Include Files

- mkl.h

Description

The routine balances a matrix $A$ by performing either or both of the following two similarity transformations:

(1) The routine first attempts to permute $A$ to block upper triangular form:

$$
P A P^T = A' = \begin{bmatrix}
A'_{11} & A'_{12} & A'_{13} \\
0 & A'_{22} & A'_{23} \\
0 & 0 & A'_{33}
\end{bmatrix}
$$

where $P$ is a permutation matrix, and $A'_{11}$ and $A'_{33}$ are upper triangular. The diagonal elements of $A'_{11}$ and $A'_{33}$ are eigenvalues of $A$. The rest of the eigenvalues of $A$ are the eigenvalues of the central diagonal block $A'_{22}$, in rows and columns $ilo$ to $ihi$. Subsequent operations to compute the eigenvalues of $A$ (or its Schur factorization) need only be applied to these rows and columns; this can save a significant amount of work if $ilo > 1$ and $ihi < n$.

If no suitable permutation exists (as is often the case), the routine sets $ilo = 1$ and $ihi = n$, and $A'_{22}$ is the whole of $A$.

(2) The routine applies a diagonal similarity transformation to $A'$, to make the rows and columns of $A'_{22}$ as close in norm as possible:

$$
A'' = D A' D^{-1} = \begin{bmatrix}
I & 0 & 0 \\
0 & D_{22} & 0 \\
0 & 0 & I
\end{bmatrix} \times \begin{bmatrix}
A'_{11} & A'_{12} & A'_{13} \\
0 & A'_{22} & A'_{23} \\
0 & 0 & A'_{33}
\end{bmatrix} \times \begin{bmatrix}
I & 0 & 0 \\
0 & D_{22}^{-1} & 0 \\
0 & 0 & I
\end{bmatrix}
$$

This scaling can reduce the norm of the matrix (that is, $||A''_{22}|| < ||A'_{22}||$), and hence reduce the effect of rounding errors on the accuracy of computed eigenvalues and eigenvectors.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **job**
  Must be 'N' or 'P' or 'S' or 'B'.
  - If `job` = 'N', then `A` is neither permuted nor scaled (but `ilo`, `ihi`, and `scale` get their values).
  - If `job` = 'P', then `A` is permuted but not scaled.
  - If `job` = 'S', then `A` is scaled but not permuted.
  - If `job` = 'B', then `A` is both scaled and permuted.

- **n**
  The order of the matrix `A` (\(n \geq 0\)).

- **a**
  Array `a` (size `max(1, lda*n)`) contains the matrix `A`.

- **lda**
  The leading dimension of `a`; at least `max(1, n)`.

**Output Parameters**

- **a**
  Overwritten by the balanced matrix (\(a\) is not referenced if `job` = 'N').

- **ilo, ihi**
  The values `ilo` and `ihi` such that on exit `a(i,j)` is zero if \(i > j\) and \(1 \leq j < ilo\) or \(ihi < j \leq n\).
  - If `job` = 'N' or 'S', then `ilo = 1` and `ihi = n`.

- **scale**
  Array, size at least `max(1, n)`.
  Contains details of the permutations and scaling factors.
  More precisely, if \(p_j\) is the index of the row and column interchanged with row and column \(j\), and \(d_j\) is the scaling factor used to balance row and column \(j\), then
  \[
  \text{scale}[j-1] = p_j \text{ for } j = 1, 2, \ldots, \text{ilo-1, ihi+1, \ldots, n}; \\
  \text{scale}[j-1] = d_j \text{ for } j = \text{ilo, ilo+1, \ldots, ihi}.
  \]
  The order in which the interchanges are made is \(n\) to \(ihi+1\), then \(1\) to \(ilo\).

**Return Values**

This function returns a value `info`.

- If `info=0`, the execution is successful.
- If `info = -i`, the \(i\)-th parameter had an illegal value.

**Application Notes**

The errors are negligible, compared with those in subsequent computations.

- If the matrix `A` is balanced by this routine, then any eigenvectors computed subsequently are eigenvectors of the matrix `A''` and hence you must call `gebak` to transform them back to eigenvectors of `A`.
- If the Schur vectors of `A` are required, do not call this routine with `job` = 'S' or 'B', because then the balancing transformation is not orthogonal (not unitary for complex flavors).
If you call this routine with \(\text{job} = 'P'\), then any Schur vectors computed subsequently are Schur vectors of the matrix \(A''\), and you need to call \texttt{gebak} (with \texttt{side} = 'R') to transform them back to Schur vectors of \(A\).

The total number of floating-point operations is proportional to \(n^2\).

\texttt{?gebak}

Transforms eigenvectors of a balanced matrix to those of the original nonsymmetric matrix.

**Syntax**

```c

lapack_int LAPACKE_sgebak( int matrix_layout, char job, char side, lapack_int n,
                        lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, float* v, lapack_int ldv );

lapack_int LAPACKE_dgebak( int matrix_layout, char job, char side, lapack_int n,
                        lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m, double* v, lapack_int ldv );

lapack_int LAPACKE_cgebak( int matrix_layout, char job, char side, lapack_int n,
                        lapack_int ilo, lapack_int ihi, const float* scale, lapack_int m, lapack_complex_float* v, lapack_int ldv );

lapack_int LAPACKE_zgebak( int matrix_layout, char job, char side, lapack_int n,
                        lapack_int ilo, lapack_int ihi, const double* scale, lapack_int m, lapack_complex_double* v, lapack_int ldv );
```

**Include Files**

- \texttt{mkl.h}

**Description**

The routine is intended to be used after a matrix \(A\) has been balanced by a call to \texttt{?gebal}, and eigenvectors of the balanced matrix \(A''\) have subsequently been computed. For a description of balancing, see \texttt{gebal}. The balanced matrix \(A''\) is obtained as \(A'' = D*P*A*P^T \text{inv}(D)\), where \(P\) is a permutation matrix and \(D\) is a diagonal scaling matrix. This routine transforms the eigenvectors as follows:

if \(x\) is a right eigenvector of \(A''\), then \(P^T \text{inv}(D) * x\) is a right eigenvector of \(A\); if \(y\) is a left eigenvector of \(A''\), then \(P^*D^*y\) is a left eigenvector of \(A\).

**Input Parameters**

- **\texttt{matrix_layout}**
  Specifies whether matrix storage layout is row major (\texttt{LAPACK_ROW_MAJOR}) or column major (\texttt{LAPACK_COL_MAJOR}).

- **\texttt{job}**
  Must be 'N' or 'P' or 'S' or 'B'. The same parameter \texttt{job} as supplied to \texttt{?gebal}.

- **\texttt{side}**
  Must be 'L' or 'R'.

  If \texttt{side} = 'L', then left eigenvectors are transformed.

  If \texttt{side} = 'R', then right eigenvectors are transformed.

- **\texttt{n}**
  The number of rows of the matrix of eigenvectors (\(n \geq 0\)).

- **\texttt{ilo, ihi}**
  The values \texttt{ilo} and \texttt{ihi}, as returned by \texttt{?gebal}. (If \(n > 0\), then 1 \(\leq \text{ilo} \leq \text{ihi} \leq n\);
if \( n = 0 \), then \( ilo = 1 \) and \( ihi = 0 \).)

\textbf{scale}

Array, size at least \( \max(1, n) \).
Contains details of the permutations and/or the scaling factors used to balance the original general matrix, as returned by \texttt{?gebal}.

\textbf{m}

The number of columns of the matrix of eigenvectors \((m \geq 0)\).

\textbf{v}

Arrays:

\( v(\text{size } \max(1, ldv*n) \text{ for column major layout and } \max(1, ldv*m) \text{ for row major layout}) \) contains the matrix of left or right eigenvectors to be transformed.

\( ldv \)

The leading dimension of \( v \); at least \( \max(1, n) \) for column major layout and at least \( \max(1, m) \) for row major layout.

\textbf{Output Parameters}

\( v \)

Overwritten by the transformed eigenvectors.

\textbf{Return Values}

This function returns a value \( info \).
If \( info=0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

\textbf{Application Notes}

The errors in this routine are negligible.
The approximate number of floating-point operations is approximately proportional to \( m*n \).

\texttt{?hseqr}

\textit{Computes all eigenvalues and (optionally) the Schur factorization of a matrix reduced to Hessenberg form.}

\textbf{Syntax}

\begin{verbatim}
lapack_int LAPACKE_shseqr( int \textit{matrix_layout}, char job, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* wr, float* wi, float* 
z, lapack_int ldz );

lapack_int LAPACKE_dhseqr( int \textit{matrix_layout}, char job, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* wr, double* wi, 
double* z, lapack_int ldz );

lapack_int LAPACKE_chseqr( int \textit{matrix_layout}, char job, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh, 
lapack_complex_float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhseqr( int \textit{matrix_layout}, char job, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh, 
lapack_complex_double* w, lapack_complex_double* z, lapack_int ldz );
\end{verbatim}

\textbf{Include Files}

\begin{itemize}
  \item mkl.h
\end{itemize}
Description

The routine computes all the eigenvalues, and optionally the Schur factorization, of an upper Hessenberg matrix \( H = Z^* T^* Z^H \), where \( T \) is an upper triangular (or, for real flavors, quasi-triangular) matrix (the Schur form of \( H \)), and \( Z \) is the unitary or orthogonal matrix whose columns are the Schur vectors \( z_i \).

You can also use this routine to compute the Schur factorization of a general matrix \( A \) which has been reduced to upper Hessenberg form \( H \):

\[
A = QH^*Q^H, \quad \text{where } Q \text{ is unitary (orthogonal for real flavors)};
\]

\[
A = (QZ)^*T^*(QZ)^H.
\]

In this case, after reducing \( A \) to Hessenberg form by \( \text{gehrd} \), call \( \text{orghr} \) to form \( Q \) explicitly and then pass \( Q \) to \( \text{?hseqr} \) with \( \text{compz} = 'V' \).

You can also call \( \text{gebal} \) to balance the original matrix before reducing it to Hessenberg form by \( \text{?hseqr} \), so that the Hessenberg matrix \( H \) will have the structure:

\[
\begin{bmatrix}
H_{11} & H_{12} & H_{13} \\
0 & H_{22} & H_{23} \\
0 & 0 & H_{33}
\end{bmatrix}
\]

where \( H_{11} \) and \( H_{33} \) are upper triangular.

If so, only the central diagonal block \( H_{22} \) (in rows and columns \( ilo \) to \( ihi \)) needs to be further reduced to Schur form (the blocks \( H_{12} \) and \( H_{23} \) are also affected). Therefore the values of \( ilo \) and \( ihi \) can be supplied to \( \text{?hseqr} \) directly. Also, after calling this routine you must call \( \text{gebak} \) to permute the Schur vectors of the balanced matrix to those of the original matrix.

If \( \text{?gebal} \) has not been called, however, then \( ilo \) must be set to 1 and \( ihi \) to \( n \). Note that if the Schur factorization of \( A \) is required, \( \text{?gebal} \) must not be called with \( \text{job} = 'S' \) or \( 'B' \), because the balancing transformation is not unitary (for real flavors, it is not orthogonal).

\( \text{?hseqr} \) uses a multishift form of the upper Hessenberg QR algorithm. The Schur vectors are normalized so that \( \|z_i\|_2 = 1 \), but are determined only to within a complex factor of absolute value 1 (for the real flavors, to within a factor ±1).

Input Parameters

- **job**
  
  Must be \( 'E' \) or \( 'S' \).

  If \( \text{job} = 'E' \), then eigenvalues only are required.

  If \( \text{job} = 'S' \), then the Schur form \( T \) is required.

- **compz**
  
  Must be \( 'N' \) or \( 'I' \) or \( 'V' \).

  If \( \text{compz} = 'N' \), then no Schur vectors are computed (and the array \( z \) is not referenced).

  If \( \text{compz} = 'I' \), then the Schur vectors of \( H \) are computed (and the array \( z \) is initialized by the routine).

  If \( \text{compz} = 'V' \), then the Schur vectors of \( A \) are computed (and the array \( z \) must contain the matrix \( Q \) on entry).

- **n**

  The order of the matrix \( H (n \geq 0) \).
If $A$ has been balanced by ?gebal, then ilo and ihi must contain the values returned by ?gebal. Otherwise, ilo must be set to 1 and ihi to $n$.

Arrays:

$h$, $z$
- $h$ (size max(1, ldh*$n$)) The $n$-by-$n$ upper Hessenberg matrix $H$.
- $z$ (size max(1, ldz*$n$))
  - If compz = 'V', then $z$ must contain the matrix $Q$ from the reduction to Hessenberg form.
  - If compz = 'I', then $z$ need not be set.
  - If compz = 'N', then $z$ is not referenced.

ldh
The leading dimension of $h$; at least max(1, $n$).

ldz
The leading dimension of $z$;
- If compz = 'N', then ldz ≥ 1.
- If compz = 'V' or 'I', then ldz ≥ max(1, $n$).

Output Parameters

$w$
Array, size at least max (1, $n$). Contains the computed eigenvalues, unless info>0. The eigenvalues are stored in the same order as on the diagonal of the Schur form $T$ (if computed).

$wr$, $wi$
Arrays, size at least max (1, $n$) each.
- Contain the real and imaginary parts, respectively, of the computed eigenvalues, unless info > 0. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first. The eigenvalues are stored in the same order as on the diagonal of the Schur form $T$ (if computed).

$h$
- If info = 0 and job = 'S', $h$ contains the upper quasi-triangular matrix $T$ from the Schur decomposition (the Schur form).
- If info = 0 and job = 'E', the contents of $h$ are unspecified on exit. (The output value of $h$ when info > 0 is given under the description of info below.)

$z$
- If compz = 'V' and info = 0, then $z$ contains $Q*Z$.
- If compz = 'I' and info = 0, then $z$ contains the unitary or orthogonal matrix $Z$ of the Schur vectors of $H$.
- If compz = 'N', then $z$ is not referenced.

Return Values
This function returns a value info.
- If info = 0, the execution is successful.
- If info = -$i$, the $i$-th parameter had an illegal value.
- If info = $i$, ?hseqr failed to compute all of the eigenvalues. Elements 1,2, ..., ilo-1 and i+1, i+2, ..., $n$ of the eigenvalue arrays ($wr$ and $wi$ for real flavors and $w$ for complex flavors) contain the real and imaginary parts of those eigenvalues that have been successfully found.
If $info > 0$, and $job = 'E'$, then on exit, the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns $ilo$ through $info$ of the final output value of $H$.

If $info > 0$, and $job = 'S'$, then on exit (initial value of $H\ast U = U\ast$ (final value of $H$), where $U$ is a unitary matrix. The final value of $H$ is upper Hessenberg and triangular in rows and columns $info+1$ through $ihi$.

If $info > 0$, and $compz = 'V'$, then on exit (final value of $Z$) = (initial value of $Z\ast U$, where $U$ is the unitary matrix (regardless of the value of $job$).

If $info > 0$, and $compz = 'I'$, then on exit (final value of $Z$) = $U$, where $U$ is the unitary matrix (regardless of the value of $job$).

If $info > 0$, and $compz = 'N'$, then $Z$ is not accessed.

Application Notes

The computed Schur factorization is the exact factorization of a nearby matrix $H + E$, where $\|E\|_2 < O(\varepsilon)$ $\|H\|_2/s_i$, and $\varepsilon$ is the machine precision.

If $\lambda_i$ is an exact eigenvalue, and $\mu_i$ is the corresponding computed value, then $|\lambda_i - \mu_i| \leq c(n) \ast \varepsilon \ast \|H\|_2/s_i$, where $c(n)$ is a modestly increasing function of $n$, and $s_i$ is the reciprocal condition number of $\lambda_i$. The condition numbers $s_i$ may be computed by calling trsna.

The total number of floating-point operations depends on how rapidly the algorithm converges; typical numbers are as follows.

If only eigenvalues are computed: $7n^3$ for real flavors

$25n^3$ for complex flavors.

If the Schur form is computed: $10n^3$ for real flavors

$35n^3$ for complex flavors.

If the full Schur factorization is computed: $20n^3$ for real flavors

$70n^3$ for complex flavors.

$hsein$

Computes selected eigenvectors of an upper Hessenberg matrix that correspond to specified eigenvalues.

Syntax

```c
lapack_int LAPACKE_shsein( int matrix_layout, char side, char eigsr, char initv,
                           lapack_logical* select, lapack_int n, const float* h, lapack_int ldh, float* wr, const
                           float* wi, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int mm,
                           lapack_int* m, lapack_int* ifaili, lapack_int* ifailr );
```

```c
lapack_int LAPACKE_dhsein( int matrix_layout, char side, char eigsr, char initv,
                           lapack_logical* select, lapack_int n, const double* h, lapack_int ldh, double* wr,
                           const double* wi, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int mm,
                           lapack_int* m, lapack_int* ifaili, lapack_int* ifailr );
```

```c
lapack_int LAPACKE_chsein( int matrix_layout, char side, char eigsr, char initv, const
                           lapack_logical* select, lapack_int n, const lapack_complex_float* h, lapack_int ldh,
                           lapack_complex_float* w, lapack_complex_float* vl, lapack_int ldvl,
                           lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int* ifaili,
                           lapack_int* ifailr );
```
lapack_int LAPACKE_zhsein( int matrix_layout, char side, char eigsrc, char initv, const
lapack_logical* select, lapack_int n, const lapack_complex_double* h, lapack_int ldh,
lapack_complex_double* w, lapack_complex_double* vl, lapack_int ldvl,
lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m, lapack_int* ifaill, lapack_int* ifailr);

Include Files
• mkl.h

Description
The routine computes left and/or right eigenvectors of an upper Hessenberg matrix \( H \), corresponding to
selected eigenvalues.

The right eigenvector \( x \) and the left eigenvector \( y \), corresponding to an eigenvalue \( \lambda \), are defined by:
\[ H^*x = \lambda^*x \quad \text{and} \quad y^*H = \lambda^*y \] (or \( H^*y = \lambda^*y \)). Here \( \lambda^* \) denotes the conjugate of \( \lambda \).

The eigenvectors are computed by inverse iteration. They are scaled so that, for a real eigenvector \( x \), \( \max| x_i | = 1 \), and for a complex eigenvector, \( \max(|\Re x_i| + |\Im x_i|) = 1 \).

If \( H \) has been formed by reduction of a general matrix \( A \) to upper Hessenberg form, then eigenvectors of \( H \)
may be transformed to eigenvectors of \( A \) by \text{ormhr} or \text{unmhr}.

Input Parameters

side
Must be 'R' or 'L' or 'B'.
If \( \text{side} = 'R' \), then only right eigenvectors are computed.
If \( \text{side} = 'L' \), then only left eigenvectors are computed.
If \( \text{side} = 'B' \), then all eigenvectors are computed.

eigsrc
Must be 'Q' or 'N'.
If \( \text{eigsrc} = 'Q' \), then the eigenvalues of \( H \) were found using \text{hseqr}; thus if \( H \)
has any zero sub-diagonal elements (and so is block triangular), then the
\( j \)-th eigenvalue can be assumed to be an eigenvalue of the block containing
the \( j \)-th row/column. This property allows the routine to perform inverse
iteration on just one diagonal block. If \( \text{eigsrc} = 'N' \), then no such
assumption is made and the routine performs inverse iteration using the
whole matrix.

initv
Must be 'N' or 'U'.
If \( \text{initv} = 'N' \), then no initial estimates for the selected eigenvectors are
supplied.
If \( \text{initv} = 'U' \), then initial estimates for the selected eigenvectors are
supplied in \( vl \) and/or \( vr \).

select
Array, size at least max (1, \( n \)). Specifies which eigenvectors are to be
computed.
For real flavors:
To obtain the real eigenvector corresponding to the real eigenvalue \( \text{wr}[j] \),
set \( \text{select}[j] \) to 1
To select the complex eigenvector corresponding to the complex eigenvalue \((wr[j - 1], wi[j - 1])\) with complex conjugate \((wr[j], wi[j])\), set \(select[j - 1]\) and/or \(select[j]\) to 1; the eigenvector corresponding to the first eigenvalue in the pair is computed.

For complex flavors:

To select the eigenvector corresponding to the eigenvalue \(w[j]\), set \(select[j]\) to 1.

The order of the matrix \(H (n \geq 0)\).

**Arrays:**

- \(h\) (size \(max(1, ldh*n)\)) The \(n\)-by-\(n\) upper Hessenberg matrix \(H\). If an \textit{NAN} value is detected in \(h\), the routine returns with \textit{info} = -6.
- \(vl\) (size \(max(1, ldvl*mm)\) for column major layout and \(max(1, ldvl*n)\) for row major layout)
  - If \(initv = 'V'\) and \(side = 'L'\) or 'B', then \(vl\) must contain starting vectors for inverse iteration for the left eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.
  - If \(initv = 'N'\), then \(vl\) need not be set.
  - The array \(vl\) is not referenced if \(side = 'R'\).
- \(vr\) (size \(max(1, ldvr*mm)\) for column major layout and \(max(1, ldvr*n)\) for row major layout)
  - If \(initv = 'V'\) and \(side = 'R'\) or 'B', then \(vr\) must contain starting vectors for inverse iteration for the right eigenvectors. Each starting vector must be stored in the same column or columns as will be used to store the corresponding eigenvector.
  - If \(initv = 'N'\), then \(vr\) need not be set.
  - The array \(vr\) is not referenced if \(side = 'L'\).

- \(ldh\)
  - The leading dimension of \(h\); at least \(max(1, n)\).

- \(w\)
  - Array, size at least \(max(1, n)\).
  - Contains the eigenvalues of the matrix \(H\).
  - If \(eigsrc = 'Q'\), the array must be exactly as returned by \textit{?hseqr}.

- \(wr, wi\)
  - Arrays, size at least \(max(1, n)\) each.
  - Contain the real and imaginary parts, respectively, of the eigenvalues of the matrix \(H\). Complex conjugate pairs of values must be stored in consecutive elements of the arrays. If \(eigsrc = 'Q'\), the arrays must be exactly as returned by \textit{?hseqr}.

- \(ldvl\)
  - The leading dimension of \(vl\).
  - If \(side = 'L'\) or 'B', \(ldvl \geq max(1, n)\) for column major layout and \(ldvl \geq max(1, mm)\) for row major layout.
  - If \(side = 'R'\), \(ldvl \geq 1\).

- \(ldvr\)
  - The leading dimension of \(vr\).
If $\text{side} = 'R' \lor 'B'$, $ldvr \geq \max(1, n)$ for column major layout and $ldvr \geq \max(1, mm)$ for row major layout.

If $\text{side} = 'L'$, $ldvr \geq 1$.

$mm$

The number of columns in $vl$ and/or $vr$.

Must be at least $m$, the actual number of columns required (see Output Parameters below).

For real flavors, $m$ is obtained by counting $1$ for each selected real eigenvector and $2$ for each selected complex eigenvector (see select).

For complex flavors, $m$ is the number of selected eigenvectors (see select).

Constraint:

$0 \leq mm \leq n$.

Output Parameters

$select$

Overwritten for real flavors only.

If a complex eigenvector was selected as specified above, then $select[j - 1]$ is set to $1$ and $select[j]$ to $0$.

$w$

The real parts of some elements of $w$ may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

$wr$

Some elements of $wr$ may be modified, as close eigenvalues are perturbed slightly in searching for independent eigenvectors.

$v1, vr$

If $\text{side} = 'L' \lor 'B'$, $vl$ contains the computed left eigenvectors (as specified by $select$).

If $\text{side} = 'R' \lor 'B'$, $vr$ contains the computed right eigenvectors (as specified by $select$).

The eigenvectors treated column-wise form a rectangular $n$-by-$mm$ matrix.

For real flavors: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by $\text{matrix_layout}$ (using either column major or row major layout).

For real flavors: the number of columns of $vl$ and/or $vr$ required to store the selected eigenvectors.

For complex flavors: the number of selected eigenvectors.

$ifaill, ifailr$

Arrays, size at least $\max(1, mm)$ each.

$ifaill[i - 1] = 0$ if the $i$th column of $vl$ converged;

$ifaill[i - 1] = j > 0$ if the eigenvector stored in the $i$-th column of $vl$ (corresponding to the $j$th eigenvalue) failed to converge.

$ifailr[i - 1] = 0$ if the $i$th column of $vr$ converged;
ifailr[i - 1] = j > 0 if the eigenvector stored in the i-th column of vr (corresponding to the jth eigenvalue) failed to converge.

For real flavors: if the ith and (i+1)th columns of vl contain a selected complex eigenvector, then ifaill[i - 1] and ifailr[i] are set to the same value. A similar rule holds for vr and ifailr.

The array ifaill is not referenced if side = 'R'. The array ifailr is not referenced if side = 'L'.

Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info > 0, then i eigenvectors (as indicated by the parameters ifaill and/or ifailr above) failed to converge. The corresponding columns of vl and/or vr contain no useful information.

Application Notes

Each computed right eigenvector xi is the exact eigenvector of a nearby matrix A + Ei, such that ||Ei|| < O(ε)||A||. Hence the residual is small:

||Ax_i - \lambda_i x_i|| = O(ε)||A||.

However, eigenvectors corresponding to close or coincident eigenvalues may not accurately span the relevant subspaces.

Similar remarks apply to computed left eigenvectors.

?trevc

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by ?hseqr.

Syntax

```c
lapack_int LAPACKE_strevc( int matrix_layout, char side, char howmny, lapack_logical* select, lapack_int n, const float* t, lapack_int ldt, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

```c
lapack_int LAPACKE_dtrevc( int matrix_layout, char side, char howmny, lapack_logical* select, lapack_int n, const double* t, lapack_int ldt, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

```c
lapack_int LAPACKE_ctrevc( int matrix_layout, char side, char howmny, const lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt, lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

```c
lapack_int LAPACKE_ztrevc( int matrix_layout, char side, char howmny, const lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m );
```

Include Files

- mkl.h
Description

The routine computes some or all of the right and/or left eigenvectors of an upper triangular matrix $T$ (or, for real flavors, an upper quasi-triangular matrix $T$). Matrices of this type are produced by the Schur factorization of a general matrix: $A = Q^* T Q^H$, as computed by hseqr.

The right eigenvector $x$ and the left eigenvector $y$ of $T$ corresponding to an eigenvalue $w$, are defined by:

$$T^* x = w^* x, \quad y^H T = w^* y^H,$$

where $y^H$ denotes the conjugate transpose of $y$.

The eigenvalues are not input to this routine, but are read directly from the diagonal blocks of $T$.

This routine returns the matrices $X$ and/or $Y$ of right and left eigenvectors of $T$, or the products $Q^* X$ and/or $Q^* Y$, where $Q$ is an input matrix.

If $Q$ is the orthogonal/unitary factor that reduces a matrix $A$ to Schur form $T$, then $Q^* X$ and $Q^* Y$ are the matrices of right and left eigenvectors of $A$.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **side**: Must be 'R' or 'L' or 'B'.
  - If $side = 'R'$, then only right eigenvectors are computed.
  - If $side = 'L'$, then only left eigenvectors are computed.
  - If $side = 'B'$, then all eigenvectors are computed.
- **howmny**: Must be 'A' or 'B' or 'S'.
  - If $howmny = 'A'$, then all eigenvectors (as specified by $side$) are computed.
  - If $howmny = 'B'$, then all eigenvectors (as specified by $side$) are computed and backtransformed by the matrices supplied in $vl$ and $vr$.
  - If $howmny = 'S'$, then selected eigenvectors (as specified by $side$ and $select$) are computed.
- **select**: Array, size at least max (1, $n$).
  - If $howmny = 'S'$, $select$ specifies which eigenvectors are to be computed.
  - If $howmny = 'A'$ or 'B', $select$ is not referenced.

  For real flavors:
  - If $omega[j]$ is a real eigenvalue, the corresponding real eigenvector is computed if $select[j]$ is 1.
  - If $omega[j - 1]$ and $omega[j]$ are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either $select[j - 1]$ or $select[j]$ is 1, and on exit $select[j - 1]$ is set to 1 and $select[j]$ is set to 0.

  For complex flavors:
  - The eigenvector corresponding to the $j$-th eigenvalue is computed if $select[j - 1]$ is 1.
- **$n$**: The order of the matrix $T$ ($n \geq 0$).
Arrays:

t (size \(\max(1, ldt \times n)\)) contains the \(n\)-by-\(n\) matrix \(T\) in Schur canonical form. For complex flavors \texttt{ctrevc} and \texttt{ztrevc}, contains the upper triangular matrix \(T\).

\(vl\) (size \(\max(1, ldvl \times mm)\) for column major layout and \(\max(1, ldvl \times n)\) for row major layout)

If \texttt{howmny} = 'B' and \texttt{side} = 'L' or 'B', then \(vl\) must contain an \(n\)-by-\(n\) matrix \(Q\) (usually the matrix of Schur vectors returned by \texttt{?hseqr}).

If \texttt{howmny} = 'A' or 'S', then \(vl\) need not be set.

The array \(vl\) is not referenced if \texttt{side} = 'R'.

\(vr\) (size \(\max(1, ldvr \times mm)\) for column major layout and \(\max(1, ldvr \times n)\) for row major layout)

If \texttt{howmny} = 'B' and \texttt{side} = 'R' or 'B', then \(vr\) must contain an \(n\)-by-\(n\) matrix \(Q\) (usually the matrix of Schur vectors returned by \texttt{?hseqr}).

If \texttt{howmny} = 'A' or 'S', then \(vr\) need not be set.

The array \(vr\) is not referenced if \texttt{side} = 'L'.

\(ldt\)

The leading dimension of \(t\); at least \(\max(1, n)\).

\(ldvl\)

The leading dimension of \(vl\).

If \texttt{side} = 'L' or 'B', \(ldvl \geq n\).

If \texttt{side} = 'R', \(ldvl \geq 1\).

\(ldvr\)

The leading dimension of \(vr\).

If \texttt{side} = 'R' or 'B', \(ldvr \geq n\).

If \texttt{side} = 'L', \(ldvr \geq 1\).

\(mm\)

The number of columns in the arrays \(vl\) and/or \(vr\). Must be at least \(m\) (the precise number of columns required).

If \texttt{howmny} = 'A' or 'B', \(mm = n\).

If \texttt{howmny} = 'S': for real flavors, \(mm\) is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, \(mm\) is the number of selected eigenvectors (see \texttt{select}).

Constraint: \(0 \leq mm \leq n\).

Output Parameters

\(select\)

If a complex eigenvector of a real matrix was selected as specified above, then \texttt{select}[j] is set to 1 and \texttt{select}[j + 1] to 0.

\(t\)

\texttt{ctrevc/ztrevc} modify the \(t\) array, which is restored on exit.

\(vl, vr\)

If \texttt{side} = 'L' or 'B', \(vl\) contains the computed left eigenvectors (as specified by \texttt{howmny} and \texttt{select}).
If \( \text{side} = 'R' \) or \( 'B' \), \( \text{vr} \) contains the computed right eigenvectors (as specified by \( \text{howmny} \) and \( \text{select} \)).

The eigenvectors treated column-wise form a rectangular \( n \)-by-\( mm \) matrix.

For real flavors: a real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns: the first column holds the real part of the eigenvector and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by \( \text{matrix_layout} \) (using either column major or row major layout).

For complex flavors: the number of selected eigenvectors. If \( \text{howmny} = 'A' \) or \( 'B' \), \( m \) is set to \( n \).

For real flavors: the number of columns of \( \text{vl} \) and/or \( \text{vr} \) actually used to store the selected eigenvectors. If \( \text{howmny} = 'A' \) or \( 'B' \), \( m \) is set to \( n \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

Application Notes

If \( x_i \) is an exact right eigenvector and \( y_i \) is the corresponding computed eigenvector, then the angle \( \theta(y_i, x_i) \) between them is bounded as follows: \( \theta(y_i, x_i) \leq \frac{c(n)\varepsilon||T||_2}{\text{sep}_i} \) where \( \text{sep}_i \) is the reciprocal condition number of \( x_i \). The condition number \( \text{sep}_i \) may be computed by calling \( \text{?trsna} \).

\texttt{?trevc3}

Computes selected eigenvectors of an upper (quasi-) triangular matrix computed by \texttt{?hseqr} using Level 3 BLAS

Syntax

\begin{align*}
\text{call strevc3} & \left( \text{side}, \text{howmny}, \text{select}, n, t, \text{ldt}, \text{vl}, \text{ldvl}, \text{vr}, \text{ldvr}, mm, m, \text{work}, \text{lwork}, \text{info} \right) \\
\text{call dtrevc3} & \left( \text{side}, \text{howmny}, \text{select}, n, t, \text{ldt}, \text{vl}, \text{ldvl}, \text{vr}, \text{ldvr}, mm, m, \text{work}, \text{lwork}, \text{info} \right) \\
\text{call ctrevc3} & \left( \text{side}, \text{howmny}, \text{select}, n, t, \text{ldt}, \text{vl}, \text{ldvl}, \text{vr}, \text{ldvr}, mm, m, \text{work}, \text{lwork}, \text{rwork}, \text{lrwork}, \text{info} \right) \\
\text{call ztrevc3} & \left( \text{side}, \text{howmny}, \text{select}, n, t, \text{ldt}, \text{vl}, \text{ldvl}, \text{vr}, \text{ldvr}, mm, m, \text{work}, \text{lwork}, \text{rwork}, \text{lrwork}, \text{info} \right)
\end{align*}

Include Files

- \texttt{mkl.fi}
Description

This routine computes some or all of the right and left eigenvectors of an upper triangular matrix \( T \) (or, for real flavors, an upper quasi-triangular matrix \( T \)) using Level 3 BLAS. Matrices of this type are produced by the Schur factorization of a general matrix: \( A = Q^* T Q \), as computed by hseqr.

The right eigenvector \( x \) and the left eigenvector \( y \) of \( T \) corresponding to an eigenvalue \( w \) are defined by the following:

\[
T^* x = w x, \quad y^* T = w y^*
\]

where \( y^* \) denotes the conjugate transpose of \( y \).

The eigenvalues are not passed to this routine but are read directly from the diagonal blocks of \( T \).

This routine returns one or both of the matrices \( X \) and \( Y \) of the right and left eigenvectors of \( T \), or one or both of the products \( Q^* X \) and \( Q^* Y \), where \( Q \) is an input matrix.

If \( Q \) is the orthogonal/unitary factor that reduces a matrix \( A \) to Schur form \( T \), then \( Q^* X \) and \( Q^* Y \) are the matrices of the right and left eigenvectors of \( A \).

Input Parameters

\( side \) \hspace{1cm} \text{CHARACTER*1}

Must be 'R', 'L', or 'B'.

- If \( side = 'R' \), only right eigenvectors are computed.
- If \( side = 'L' \), only left eigenvectors are computed.
- If \( side = 'B' \), all eigenvectors are computed.

\( howmny \) \hspace{1cm} \text{CHARACTER*1}

Must be 'A', 'B', or 'S'.

- If \( howmny = 'A' \), all eigenvectors (as specified by \( side \)) are computed.
- If \( howmny = 'B' \), all eigenvectors (as specified by \( side \)) are computed and back-transformed by the matrices supplied in \( vl \) and \( vr \).
- If \( howmny = 'S' \), selected eigenvectors (as specified by \( side \) and \( select \)) are computed.

\( select \) \hspace{1cm} \text{Array with a size of at least max (1, n)}

If \( howmny = 'S' \), \( select \) specifies which eigenvectors are to be computed. If \( howmny = 'A' \) or \( howmny = 'B' \), \( select \) is not referenced.

For real flavors:

- If \( \omega(j) \) is a real eigenvalue and \( select(j) \) is .TRUE., the corresponding real eigenvector is computed.
- If \( \omega(j) \) and \( \omega(j + 1) \) are the real and imaginary parts of a complex eigenvalue and either \( select(j) \) or \( select(j + 1) \) is .TRUE., the corresponding complex eigenvector is computed, and on exit \( select(j) \) is set to .TRUE. and \( select(j + 1) \) is set to .FALSE..

For complex flavors:
• If `select(j)` is `.TRUE.`, the eigenvector corresponding to the \( j \)th eigenvalue is computed.

\[ n \]

INTEGER

The order of the matrix \( T \) \((n \geq 0)\).

\[ t, vl, vr, work \]

• REAL for `strevc3`
• DOUBLE PRECISION for `dtrevc3`
• COMPLEX for `ctrevc3`
• DOUBLE COMPLEX for `ztrevc3`

Arrays:

• \( t(ldt,* \)\) contains the \( n \)-by-\( n \) matrix \( T \) in Schur canonical form. For complex flavors `ctrevc3` and `ztrevc3`, the array contains the upper triangular matrix \( T \).

The second dimension of \( t \) must be at least \( \max(1, n) \).

• \( vl(ldvl,*) \)

If `howmny` = 'B' and `side` = 'L' or 'B', then \( vl \) must contain an \( n \)-by-\( n \) matrix \( Q \) (usually the matrix of Schur vectors returned by `?hseqr`).

If `howmny` = 'A' or 'S', \( vl \) need not be set.

The second dimension of \( vl \) must be at least \( \max(1, mm) \) if `side` = 'L' or 'B', and at least 1 if `side` = 'R'.

The array \( vl \) is not referenced if `side` = 'R'.

• \( vr(ldvr,*) \)

If `howmny` = 'B' and `side` = 'R' or 'B', \( vr \) must contain an \( n \)-by-\( n \) matrix \( Q \) (usually the matrix of Schur vectors returned by `?hseqr`).

If `howmny` = 'A' or 'S', \( vr \) need not be set.

The second dimension of \( vr \) must be at least \( \max(1, mm) \) if `side` = 'R' or 'B', and at least 1 if `side` = 'L'.

The array \( vr \) is not referenced if `side` = 'L'.

• \( work(*) \) is a workspace array, and its dimension is \( \max(1, lwork) \).

\[ lwork \]

INTEGER

The size of the work array. Must be at least \( \max(1, 3*n) \) for real flavors, and at least \( \max(1, 2*n) \) for complex flavors.

If `lwork` = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the work array, and no error message related to `lwork` is issued by `xerbla`. For details, see "Application Notes" below.

\[ ldt \]

INTEGER

The leading dimension of \( t \). It is at least \( \max(1, n) \).

\[ ldvl \]

INTEGER
The leading dimension of vl.

- If side = 'L' or 'B', ldvl≥n.
- If side = 'R', ldvl≥ 1.

ldvr

INTEGER

The leading dimension of vr.

- If side = 'R' or 'B', ldvr≥n.
- If side = 'L', ldvr≥ 1.

mm

INTEGER

The number of columns in one or both of the arrays vl and vr. Must be at least m (the precise number of columns required).

- If howmny = 'A' or 'B', mm = n.
- If howmny = 'S': for real flavors, mm is obtained by counting 1 for each selected real eigenvector and 2 for each selected complex eigenvector; for complex flavors, mm is the number of selected eigenvectors (see select).

Constraint: 0 ≤ mm ≤ n.

rwork

- REAL for ctrrvec3
- DOUBLE PRECISION for zttrvec3

The workspace array is used in complex flavors only. Its dimension is max (1, lrwork).

lrwork

INTEGER

The size of the rwork array. It must be at least max(1, n).

If lrwork = -1, a workspace query is assumed; the routine calculates only the optimal size of the work array and returns this value as the first entry of the rwork array, and no error message related to lrwork is issued by xerbla. For details, see "Application Notes" below.

Output Parameters

select

If a complex eigenvector of a real matrix was selected as specified above, then select(j) is set to .TRUE. and select(j + 1) is set to .FALSE..

t

COMPLEX for ctrrvec3
DOUBLE COMPLEX for zttrvec3

trrvec3 or zttrvec3 modifies the t(ldt,*) array, which is restored on exit.

vl, vr

If side = 'L' or 'B', vl contains the computed left eigenvectors (as specified by howmny and select).

If side = 'R' or 'B', vr contains the computed right eigenvectors (as specified by howmny and select).
Treated column-wise, the eigenvectors form a rectangular $n$-by-$m$ matrix.

**For real flavors** A real eigenvector corresponding to a real eigenvalue occupies one column of the matrix; a complex eigenvector corresponding to a complex eigenvalue occupies two columns. The first column holds the real part of the eigenvector, and the second column holds the imaginary part of the eigenvector. The matrix is stored in a one-dimensional array as described by matrix_layout (using either column major or row major layout).

$m$

**For complex flavors** The number of selected eigenvectors. If howmny = 'A' or 'B', $m$ is set to $n$.

**For real flavors** The number of columns of one or both of $vl$ and $vr$ actually used to store the selected eigenvectors. If howmny = 'A' or 'B', $m$ is set to $n$.

work(1)

On exit, if info = 0, work(1) returns the required optimal size of lwork.

rwork(1)

On exit, if info = 0, then rwork(1) returns the required optimal size of lrwork.

info

INTEGER

If info = 0, the execution is successful.
If info = -i, the $i$th parameter contained an illegal value.

**Application Notes**

If $xi$ is an exact right eigenvector and $yi$ is the corresponding computed eigenvector, the angle $\theta(yi, xi)$ between them is bounded as follows:

$$\theta(yi, xi) \leq \frac{(c(n)\varepsilon||T||_2)/sepi}{sepi}$$

where $sepi$ is the reciprocal condition number of $xi$. You can compute the condition number $sepi$ by calling ?trsna.

**See Also**

Matrix Storage Schemes

?trsna

Estimates condition numbers for specified eigenvalues and right eigenvectors of an upper (quasi-) triangular matrix.
Syntax

```c
lapack_int LAPACKE_strsna( int matrix_layout, char job, char howmny, const
  lapack_logical* select, lapack_int n, const float* t, lapack_int ldt, const float* vl,
  lapack_int ldvl, const float* vr, lapack_int ldvr, float* s, float* sep, lapack_int mm,
  lapack_int* m );

lapack_int LAPACKE_dtrsna( int matrix_layout, char job, char howmny, const
  lapack_logical* select, lapack_int n, const double* t, lapack_int ldt, const double* vl,
  lapack_int ldvl, const double* vr, lapack_int ldvr, double* s, double* sep,
  lapack_int mm, lapack_int* m );

lapack_int LAPACKE_ctrsna( int matrix_layout, char job, char howmny, const
  lapack_logical* select, lapack_int n, const lapack_complex_float* t,
  lapack_int ldt, const lapack_complex_float* vl, lapack_int ldvl, const lapack_complex_float* vr,
  lapack_int ldvr, float* s, float* sep, lapack_int mm, lapack_int* m );

lapack_int LAPACKE_ztrsna( int matrix_layout, char job, char howmny, const
  lapack_logical* select, lapack_int n, const lapack_complex_double* t,
  lapack_int ldt, const lapack_complex_double* vl, lapack_int ldvl, const lapack_complex_double* vr,
  lapack_int ldvr, double* s, double* sep, lapack_int mm, lapack_int* m );
```

Include Files

- mkl.h

Description

The routine estimates condition numbers for specified eigenvalues and/or right eigenvectors of an upper triangular matrix $T$ (or, for real flavors, upper quasi-triangular matrix $T$ in canonical Schur form). These are the same as the condition numbers of the eigenvalues and right eigenvectors of an original matrix $A = Z T Z^H$ (with unitary or, for real flavors, orthogonal $Z$), from which $T$ may have been derived.

The routine computes the reciprocal of the condition number of an eigenvalue $\lambda_i$ as $s_i = \frac{|v^T u|}{\|u\| \|E\| \|v\| \|E\|}$ for real flavors and $s_i = \frac{|v^H u|}{\|u\| \|E\| \|v\| \|E\|}$ for complex flavors,

where:

- $u$ and $v$ are the right and left eigenvectors of $T$, respectively, corresponding to $\lambda_i$.
- $v^T$/$v^H$ denote transpose/conjugate transpose of $v$, respectively.

This reciprocal condition number always lies between zero (ill-conditioned) and one (well-conditioned).

An approximate error estimate for a computed eigenvalue $\lambda_i$ is then given by $\varepsilon \|T\| / s_i$, where $\varepsilon$ is the machine precision.

To estimate the reciprocal of the condition number of the right eigenvector corresponding to $\lambda_i$, the routine first calls `trexc` to reorder the diagonal elements of matrix $T$ so that $\lambda_i$ is in the leading position:

$$
T = Q \begin{bmatrix} \lambda_i & C^H \\ 0 & T_{22} \end{bmatrix} Q^H
$$

The reciprocal condition number of the eigenvector is then estimated as $sep_i$, the smallest singular value of the matrix $T_{22} - \lambda_i^* I$.

An approximate error estimate for a computed right eigenvector $u$ corresponding to $\lambda_i$ is then given by $\varepsilon \|T\| / sep_i$. 

Input Parameters

`matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

`job` Must be 'E' or 'V' or 'B'.

If `job` = 'E', then condition numbers for eigenvalues only are computed.

If `job` = 'V', then condition numbers for eigenvectors only are computed.

If `job` = 'B', then condition numbers for both eigenvalues and eigenvectors are computed.

`howmny` Must be 'A' or 'S'.

If `howmny` = 'A', then the condition numbers for all eigenpairs are computed.

If `howmny` = 'S', then condition numbers for selected eigenpairs (as specified by `select`) are computed.

`select` Array, size at least max (1, n) if `howmny` = 'S' and at least 1 otherwise.

Specifies the eigenpairs for which condition numbers are to be computed if `howmny` = 'S'.

For real flavors:
To select condition numbers for the eigenpair corresponding to the real eigenvalue \( \lambda_j \), `select[j]` must be set 1;

to select condition numbers for the eigenpair corresponding to a complex conjugate pair of eigenvalues \( \lambda_j \) and \( \lambda_{j+1} \), `select[j - 1]` and/or `select[j]` must be set 1.

For complex flavors
To select condition numbers for the eigenpair corresponding to the eigenvalue \( \lambda_j \), `select[j]` must be set 1; `select` is not referenced if `howmny` = 'A'.

`n` The order of the matrix \( T \) (\( n \geq 0 \)).

`t, vl, vr` Arrays:

\( t \) (size max(1, \( ldt \times n \))) contains the \( n \)-by-\( n \) matrix \( T \).

\( vl \) (size max(1, \( ldvl \times mm \)) for column major layout and max(1, \( ldvl \times n \)) for row major layout)

If `job` = 'E' or 'B', then \( vl \) must contain the left eigenvectors of \( T \) (or of any matrix \( Q^* T Q \) with \( Q \) unitary or orthogonal) corresponding to the eigenpairs specified by `howmny` and `select`. The eigenvectors must be stored in consecutive columns of \( vl \), as returned by `trevc` or `hsein`.

The array \( vl \) is not referenced if `job` = 'V'.

\( vr \) (size max(1, \( ldvr \times mm \)) for column major layout and max(1, \( ldvr \times n \)) for row major layout)
If \( \text{job} = 'E' \) or 'B', then \( \mathbf{v} \) must contain the right eigenvectors of \( T \) (or of any matrix \( Q^{\dagger}TQ^{\dagger} \) with \( Q \) unitary or orthogonal) corresponding to the eigenpairs specified by \( \text{howmny} \) and \( \text{select} \). The eigenvectors must be stored in consecutive columns of \( \mathbf{v} \), as returned by \text{trevc} or \text{hsein}.

The array \( \mathbf{v} \) is not referenced if \( \text{job} = 'V' \).

\( \text{ldt} \)

The leading dimension of \( t \); at least \( \max(1, n) \).

\( \text{ldvl} \)

The leading dimension of \( \mathbf{v} \).

If \( \text{job} = 'E' \) or 'B', \( \text{ldvl} \geq \max(1, n) \) for column major layout and \( \text{ldvl} \geq \max(1, mm) \) for row major layout.

If \( \text{job} = 'V' \), \( \text{ldvl} \geq 1 \).

\( \text{ldvr} \)

The leading dimension of \( \mathbf{v} \).

If \( \text{job} = 'E' \) or 'B', \( \text{ldvr} \geq \max(1, n) \) for column major layout and \( \text{ldvr} \geq \max(1, mm) \) for row major layout.

If \( \text{job} = 'R' \), \( \text{ldvr} \geq 1 \).

\( mm \)

The number of elements in the arrays \( \mathbf{s} \) and \( \mathbf{sep} \), and the number of columns in \( \mathbf{v} \) and \( \mathbf{v} \) (if used). Must be at least \( m \) (the precise number required).

If \( \text{howmny} = 'A' \), \( mm = n \);

if \( \text{howmny} = 'S' \), for real flavors \( mm \) is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues.

for complex flavors \( mm \) is the number of selected eigenpairs (see \( \text{select} \)).

Constraint:

\( 0 \leq mm \leq n \).

Output Parameters

\( s \)

Array, size at least \( \max(1, mm) \) if \( \text{job} = 'E' \) or 'B' and at least 1 if \( \text{job} = 'V' \).

Contains the reciprocal condition numbers of the selected eigenvalues if \( \text{job} = 'E' \) or 'B', stored in consecutive elements of the array. Thus \( s[j - 1] \), \( \mathbf{sep}[j - 1] \) and the \( j \)-th columns of \( \mathbf{v} \) and \( \mathbf{v} \) all correspond to the same eigenpair (but not in general the \( j \)-th eigenpair unless all eigenpairs have been selected).

For real flavors: for a complex conjugate pair of eigenvalues, two consecutive elements of \( s \) are set to the same value. The array \( s \) is not referenced if \( \text{job} = 'V' \).

\( \text{sep} \)

Array, size at least \( \max(1, mm) \) if \( \text{job} = 'V' \) or 'B' and at least 1 if \( \text{job} = 'E' \). Contains the estimated reciprocal condition numbers of the selected right eigenvectors if \( \text{job} = 'V' \) or 'B', stored in consecutive elements of the array.
For real flavors: for a complex eigenvector, two consecutive elements of sep are set to the same value; if the eigenvalues cannot be reordered to compute sep[j - 1], then sep[j - 1] is set to zero; this can only occur when the true value would be very small anyway. The array sep is not referenced if job = 'E'.

For complex flavors: the number of selected eigenpairs.
If howmny = 'A', m is set to n.

For real flavors: the number of elements of s and/or sep actually used to store the estimated condition numbers.
If howmny = 'A', m is set to n.

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

Application Notes
The computed values sep_i may overestimate the true value, but seldom by a factor of more than 3.

?trexc
Reorders the Schur factorization of a general matrix.

Syntax
lapack_int LAPACKE_strexc( int matrix_layout, char compq, lapack_int n, float* t, lapack_int ldt, float* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );
lapack_int LAPACKE_dtrexc( int matrix_layout, char compq, lapack_int n, double* t, lapack_int ldt, double* q, lapack_int ldq, lapack_int* ifst, lapack_int* ilst );
lapack_int LAPACKE_ctrexc( int matrix_layout, char compq, lapack_int n, lapack_complex_float* t, lapack_int ldt, lapack_complex_float* q, lapack_int ldq, lapack_int ifst, lapack_int ilst );
lapack_int LAPACKE_ztrexc( int matrix_layout, char compq, lapack_int n, lapack_complex_double* t, lapack_int ldt, lapack_complex_double* q, lapack_int ldq, lapack_int ifst, lapack_int ilst );

Include Files
• mkl.h

Description
The routine reorders the Schur factorization of a general matrix A = Q*T*Q^H, so that the diagonal element or block of T with row index ifst is moved to row ilst.
The reordered Schur form S is computed by an unitary (or, for real flavors, orthogonal) similarity transformation: S = Z^H*T*Z. Optionally the updated matrix P of Schur vectors is computed as P = Q*Z, giving A = P*S*P^H.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **compq**
  Must be 'V' or 'N'.
  - If `compq` = 'V', then the Schur vectors (Q) are updated.
  - If `compq` = 'N', then no Schur vectors are updated.

- **n**
  The order of the matrix T (n ≥ 0).

- **t, q**
  Arrays:
  - `t` (size max(1, ldt*n)) contains the n-by-n matrix T.
  - `q` (size max(1, ldq*n))
    - If `compq` = 'V', then `q` must contain Q (Schur vectors).
    - If `compq` = 'N', then `q` is not referenced.

- **ldt**
  The leading dimension of `t`; at least max(1, n).

- **ldq**
  The leading dimension of `q`;
  - If `compq` = 'N', then ldq ≥ 1.
  - If `compq` = 'V', then ldq ≥ max(1, n).

- **ifst, ilst**
  1 ≤ ifst ≤ n; 1 ≤ ilst ≤ n.
  Must specify the reordering of the diagonal elements (or blocks, which is possible for real flavors) of the matrix T. The element (or block) with row index `ifst` is moved to row `ilst` by a sequence of exchanges between adjacent elements (or blocks).

**Output Parameters**

- **t**
  Overwritten by the updated matrix S.

- **q**
  If `compq` = 'V', `q` contains the updated matrix of Schur vectors.

- **ifst, ilst**
  Overwritten for real flavors only.
  - If `ifst` pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; `ilst` always points to the first row of the block in its final position (which may differ from its input value by ±1).

**Return Values**

This function returns a value `info`.

- If `info`=0, the execution is successful.
- If `info` = -i, the i-th parameter had an illegal value.

**Application Notes**

The computed matrix S is exactly similar to a matrix $T+E$, where $||E||_2 = O(\varepsilon) * ||T||_2$, and $\varepsilon$ is the machine precision.
Note that if a 2 by 2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2 by 2 block to break into two 1 by 1 blocks, that is, for a pair of complex eigenvalues to become purely real.

The approximate number of floating-point operations is

for real flavors:

\[ 6n(\text{ifst-ilst}) \text{ if compq} = 'N'; \]
\[ 12n(\text{ifst-ilst}) \text{ if compq} = 'V'; \]

for complex flavors:

\[ 20n(\text{ifst-ilst}) \text{ if compq} = 'N'; \]
\[ 40n(\text{ifst-ilst}) \text{ if compq} = 'V'. \]

trsen
Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers for the selected cluster of eigenvalues and respective invariant subspace.

Syntax

```
lapack_int LAPACKE_strsen( int matrix_layout, char job, char compq, const lapack_logical* select, lapack_int n, float* t, lapack_int ldt, float* q, lapack_int ldq, float* wr, float* wi, lapack_int* m, float* s, float* sep );
lapack_int LAPACKE_dtrsen( int matrix_layout, char job, char compq, const lapack_logical* select, lapack_int n, double* t, lapack_int ldt, double* q, lapack_int ldq, double* wr, double* wi, lapack_int* m, double* s, double* sep );
lapack_int LAPACKE_ctrsen( int matrix_layout, char job, char compq, const lapack_logical* select, lapack_int n, lapack_complex_float* t, lapack_int ldt, lapack_complex_float* q, lapack_int ldq, lapack_complex_float* w, lapack_int* m, float* s, float* sep );
lapack_int LAPACKE_ztrsen( int matrix_layout, char job, char compq, const lapack_logical* select, lapack_int n, lapack_complex_double* t, lapack_int ldt, lapack_complex_double* q, lapack_int ldq, lapack_complex_double* w, lapack_int* m, double* s, double* sep );
```

Include Files

- mkl.h

Description

The routine reorders the Schur factorization of a general matrix \( A = Q^*T^*Q^T \) (for real flavors) or \( A = Q^*T^*Q^H \) (for complex flavors) so that a selected cluster of eigenvalues appears in the leading diagonal elements (or, for real flavors, diagonal blocks) of the Schur form. The reordered Schur form \( R \) is computed by a unitary (orthogonal) similarity transformation: \( R = Z^*T^*Z \). Optionally the updated matrix \( P \) of Schur vectors is computed as \( P = Q^*Z \), giving \( A = P^*R^*P^T \).

Let

\[
R = \begin{bmatrix}
T_{11} & T_{12} \\
0 & T_{13}
\end{bmatrix}
\]
where the selected eigenvalues are precisely the eigenvalues of the leading \( m \)-by-\( m \) submatrix \( T_{11} \). Let \( P \) be correspondingly partitioned as \((Q_1Q_2)\) where \( Q_1 \) consists of the first \( m \) columns of \( Q \). Then \( A^*Q_1 = Q_1^*T_{11} \), and so the \( m \) columns of \( Q_1 \) form an orthonormal basis for the invariant subspace corresponding to the selected cluster of eigenvalues.

Optionally the routine also computes estimates of the reciprocal condition numbers of the average of the cluster of eigenvalues and of the invariant subspace.

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **job**: Must be 'N' or 'E' or 'V' or 'B'.
  - If `job` = 'N', then no condition numbers are required.
  - If `job` = 'E', then only the condition number for the cluster of eigenvalues is computed.
  - If `job` = 'V', then only the condition number for the invariant subspace is computed.
  - If `job` = 'B', then condition numbers for both the cluster and the invariant subspace are computed.

- **compq**: Must be 'V' or 'N'.
  - If `compq` = 'V', then \( Q \) of the Schur vectors is updated.
  - If `compq` = 'N', then no Schur vectors are updated.

- **select**: Array, size at least max (1, \( n \)).
  - Specifies the eigenvalues in the selected cluster. To select an eigenvalue \( \lambda_j \), `select[j]` must be 1.
  - *For real flavors*: to select a complex conjugate pair of eigenvalues \( \lambda_j \) and \( \lambda_j + 1 \) (corresponding 2 by 2 diagonal block), `select[j - 1]` and/or `select[j]` must be 1; the complex conjugate \( \lambda_j \) and \( \lambda_j + 1 \) must be either both included in the cluster or both excluded.

- **n**: The order of the matrix \( T \) (\( n \geq 0 \)).

- **t, q**: Arrays:
  - \( t \) (size max(1, \( ldt*n \))) The upper quasi-triangular \( n \)-by-\( n \) matrix \( T \), in Schur canonical form.
  - \( q \) (size max(1, \( ldq*n \)))
    - If `compq` = 'V', then \( q \) must contain the matrix \( Q \) of Schur vectors.
    - If `compq` = 'N', then \( q \) is not referenced.

- **ldt**: The leading dimension of \( t \); at least max(1, \( n \)).

- **ldq**: The leading dimension of \( q \);
  - If `compq` = 'N', then \( ldq \geq 1 \).
  - If `compq` = 'V', then \( ldq \geq \max(1, \ n) \).
Output Parameters

Overwritten by the reordered matrix $R$ in Schur canonical form with the selected eigenvalues in the leading diagonal blocks.

$q$

If $compq = 'V'$, $q$ contains the updated matrix of Schur vectors; the first $m$ columns of the $Q$ form an orthogonal basis for the specified invariant subspace.

$w$

Array, size at least $\max(1, n)$. The recorded eigenvalues of $R$. The eigenvalues are stored in the same order as on the diagonal of $R$.

$wr, wi$

Arrays, size at least $\max(1, n)$. Contain the real and imaginary parts, respectively, of the reordered eigenvalues of $R$. The eigenvalues are stored in the same order as on the diagonal of $R$. Note that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

$m$

For complex flavors: the dimension of the specified invariant subspaces, which is the same as the number of selected eigenvalues (see select).

For real flavors: the dimension of the specified invariant subspace. The value of $m$ is obtained by counting 1 for each selected real eigenvalue and 2 for each selected complex conjugate pair of eigenvalues (see select).

Constraint: $0 \leq m \leq n$.

$s$

If $job = 'E$ or 'B', $s$ is a lower bound on the reciprocal condition number of the average of the selected cluster of eigenvalues.

If $m = 0$ or $n$, then $s = 1$.

For real flavors: if $info = 1$, then $s$ is set to zero.

$s$ is not referenced if $job = 'N$ or 'V'.

$sep$

If $job = 'V$ or 'B', $sep$ is the estimated reciprocal condition number of the specified invariant subspace.

If $m = 0$ or $n$, then $sep = |T|.$

For real flavors: if $info = 1$, then $sep$ is set to zero.

$sep$ is not referenced if $job = 'N$ or 'E'.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = 1$, the reordering of $T$ failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); $T$ may have been partially reordered, and $wr$ and $wi$ contain the eigenvalues in the same order as in $T$; $s$ and $sep$ (if requested) are set to zero.

Application Notes

The computed matrix $R$ is exactly similar to a matrix $T+E$, where $||E||_2 = O(\varepsilon) * ||T||_2$, and $\varepsilon$ is the machine precision. The computed $s$ cannot underestimate the true reciprocal condition number by more than a factor of $(\min(m, n-m))^{1/2}$; $sep$ may differ from the true value by $(m^*n-m^2)^{1/2}$. The angle between the
computed invariant subspace and the true subspace is $O(\epsilon) \|A\|_2/\text{sep}$. Note that if a 2-by-2 diagonal block is involved in the re-ordering, its off-diagonal elements are in general changed; the diagonal elements and the eigenvalues of the block are unchanged unless the block is sufficiently ill-conditioned, in which case they may be noticeably altered. It is possible for a 2-by-2 block to break into two 1-by-1 blocks, that is, for a pair of complex eigenvalues to become purely real.

?trsyl
Solves Sylvester equation for real quasi-triangular or complex triangular matrices.

**Syntax**

```c
lapack_int LAPACKE_strsyl( int matrix_layout, char trana, char tranb, lapack_int isgn, lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int ldb, float* c, lapack_int ldc, float* scale );

lapack_int LAPACKE_dtrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn, lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* b, lapack_int ldb, double* c, lapack_int ldc, double* scale );

lapack_int LAPACKE_ctrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn, lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, const lapack_complex_float* b, lapack_int ldb, lapack_complex_float* c, lapack_int ldc, float* scale );

lapack_int LAPACKE_ztrsyl( int matrix_layout, char trana, char tranb, lapack_int isgn, lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, const lapack_complex_double* b, lapack_intldb, lapack_complex_double* c, lapack_int ldc, double* scale );
```

**Include Files**

- mkl.h

**Description**

The routine solves the Sylvester matrix equation $\text{op}(A)X\pm X\text{op}(B) = \alpha C$, where $\text{op}(A) = A$ or $A^\text{T}$, and the matrices $A$ and $B$ are upper triangular (or, for real flavors, upper quasi-triangular in canonical Schur form); $\alpha \leq 1$ is a scale factor determined by the routine to avoid overflow in $X$; $A$ is $m$-by-$m$, $B$ is $n$-by-$n$, and $C$ and $X$ are both $m$-by-$n$. The matrix $X$ is obtained by a straightforward process of back substitution.

The equation has a unique solution if and only if $a_i \pm b_i \neq 0$, where $\{a_i\}$ and $\{b_i\}$ are the eigenvalues of $A$ and $B$, respectively, and the sign (+ or -) is the same as that used in the equation to be solved.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **trana**
  Must be 'N' or 'T' or 'C'.
  If trana = 'N', then $\text{op}(A) = A$.
  If trana = 'T', then $\text{op}(A) = A^\text{T}$ (real flavors only).
  If trana = 'C' then $\text{op}(A) = A^H$.

- **tranb**
  Must be 'N' or 'T' or 'C'.
  If tranb = 'N', then $\text{op}(B) = B$. 
If \( tranb = 'T' \), then \( \text{op}(B) = B^T \) (real flavors only).
If \( tranb = 'C' \), then \( \text{op}(B) = B^H \).

**isgn**
Indicates the form of the Sylvester equation.
If \( isgn = +1 \), \( \text{op}(A)*X + X*\text{op}(B) = \alpha * C \).
If \( isgn = -1 \), \( \text{op}(A)*X - X*\text{op}(B) = \alpha * C \).

**m**
The order of \( A \), and the number of rows in \( X \) and \( C \) \((m \geq 0)\).

**n**
The order of \( B \), and the number of columns in \( X \) and \( C \) \((n \geq 0)\).

**a, b, c**
Arrays:
\( a \) (size \( \max(1, \text{lda}*m) \)) contains the matrix \( A \).
\( b \) (size \( \max(1, \text{ldb}*n) \)) contains the matrix \( B \).
\( c \) (size \( \max(1, \text{ldc}*n) \) for column major layout and \( \max(1, \text{ldc}*m \) for row major layout) contains the matrix \( C \).

**lda**
The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

**ldb**
The leading dimension of \( b \); at least \( \max(1, n) \).

**ldc**
The leading dimension of \( c \); at least \( \max(1, m) \) for column major layout and at least \( \max(1, n) \) for row major layout.

**Output Parameters**

**c**
Overwritten by the solution matrix \( X \).

**scale**
The value of the scale factor \( \alpha \).

**Return Values**
This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.
If \( info = 1 \), \( A \) and \( B \) have common or close eigenvalues; perturbed values were used to solve the equation.

**Application Notes**
Let \( X \) be the exact, \( Y \) the corresponding computed solution, and \( R \) the residual matrix: \( R = C - (AY \pm YB) \).
Then the residual is always small:
\[
||R||_F = O(\varepsilon) \cdot (||A||_F + ||B||_F) \cdot ||Y||_F,
\]
However, \( Y \) is not necessarily the exact solution of a slightly perturbed equation; in other words, the solution is not backwards stable.
For the forward error, the following bound holds:
\[
||Y - X||_F \leq ||R||_F / \text{sep}(A, B)
\]
but this may be a considerable overestimate. See [Golub96] for a definition of \( \text{sep}(A, B) \).
The approximate number of floating-point operations for real flavors is \( mn^* (m + n) \). For complex flavors it is 4 times greater.
Generalized Nonsymmetric Eigenvalue Problems: LAPACK Computational Routines

This section describes LAPACK routines for solving generalized nonsymmetric eigenvalue problems, reordering the generalized Schur factorization of a pair of matrices, as well as performing a number of related computational tasks.

A generalized nonsymmetric eigenvalue problem is as follows: given a pair of nonsymmetric (or non-Hermitian) \( n \times n \) matrices \( A \) and \( B \), find the generalized eigenvalues \( \lambda \) and the corresponding generalized eigenvectors \( x \) and \( y \) that satisfy the equations

\[
Ax = \lambda Bx \quad (\text{right generalized eigenvectors } x)
\]

and

\[
y^H A = \lambda y^H B \quad (\text{left generalized eigenvectors } y).
\]

Table "Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems" lists LAPACK routines used to solve the generalized nonsymmetric eigenvalue problems and the generalized Sylvester equation.

### Computational Routines for Solving Generalized Nonsymmetric Eigenvalue Problems

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gghrd</td>
<td>Reduces a pair of matrices to generalized upper Hessenberg form using orthogonal/unitary transformations.</td>
</tr>
<tr>
<td>ggbal</td>
<td>Balances a pair of general real or complex matrices.</td>
</tr>
<tr>
<td>ggbak</td>
<td>Forms the right or left eigenvectors of a generalized eigenvalue problem.</td>
</tr>
<tr>
<td>gghd3</td>
<td>Reduces a pair of matrices to generalized upper Hessenberg form.</td>
</tr>
<tr>
<td>hgeqz</td>
<td>Implements the QZ method for finding the generalized eigenvalues of the matrix pair ((H,T)).</td>
</tr>
<tr>
<td>tgevc</td>
<td>Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices</td>
</tr>
<tr>
<td>tgexc</td>
<td>Reorders the generalized Schur decomposition of a pair of matrices ((A,B)) so that one diagonal block of ((A,B)) moves to another row index.</td>
</tr>
<tr>
<td>tgsen</td>
<td>Reorders the generalized Schur decomposition of a pair of matrices ((A,B)) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of ((A,B)).</td>
</tr>
<tr>
<td>tgsyl</td>
<td>Solves the generalized Sylvester equation.</td>
</tr>
<tr>
<td>tgsyl</td>
<td>Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.</td>
</tr>
</tbody>
</table>

```c
lapack_int LAPACKE_sgghrd (int matrix_layout, char compq, char compz, lapack_int n, lapack_int ilo, lapack_int ihi, float* a, lapack_int lda, float* b, lapack_int ldb, float* q, lapack_int ldq, float* z, lapack_int ldz);

lapack_int LAPACKE_dgghrd (int matrix_layout, char compq, char compz, lapack_int n, lapack_int ilo, lapack_int ihi, double* a, lapack_int lda, double* b, lapack_int ldb, double* q, lapack_int ldq, double* z, lapack_int ldz);
```
lapack_int LAPACKE_cgghrd (int matrix_layout, char compq, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, lapack_complex_float* a, lapack_int lda, 
lapack_complex_float* b, lapack_int ldb, lapack_complex_float* q, lapack_int ldq, 
lapack_complex_float* z, lapack_int ldz);

lapack_int LAPACKE_zgghrd (int matrix_layout, char compq, char compz, lapack_int n, 
lapack_int ilo, lapack_int ihi, lapack_complex_double* a, lapack_int lda, 
lapack_complex_double* b, lapack_int ldb, lapack_complex_double* q, lapack_int ldq, 
lapack_complex_double* z, lapack_int ldz);

Include Files

- mkl.h

Description

The routine reduces a pair of real/complex matrices \((A, B)\) to generalized upper Hessenberg form using orthogonal/unitary transformations, where \(A\) is a general matrix and \(B\) is upper triangular. The form of the generalized eigenvalue problem is \(A x = \lambda B x\), and \(B\) is typically made upper triangular by computing its QR factorization and moving the orthogonal matrix \(Q\) to the left side of the equation.

This routine simultaneously reduces \(A\) to a Hessenberg matrix \(H:\)
\[ Q^H A Z = H \]
and transforms \(B\) to another upper triangular matrix \(T:\)
\[ Q^H B Z = T \]
in order to reduce the problem to its standard form \(H^* y = \lambda^* T^* y\), where \(y = Z^H x\).

The orthogonal/unitary matrices \(Q\) and \(Z\) are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices \(Q_1\) and \(Z_1\), so that
\[
Q_1^* A Z_1^H = (Q_1^* Q)^* (Z_1^* Z)^H \\
Q_1^* B Z_1^H = (Q_1^* Q)^* (Z_1^* Z)^H 
\]

If \(Q_1\) is the orthogonal/unitary matrix from the QR factorization of \(B\) in the original equation \(A x = \lambda B x\), then the routine \(gghrd\) reduces the original problem to generalized Hessenberg form.

Input Parameters

matrix_layout

- Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

compq

- Must be 'N', 'I', or 'V'.
- If compq = 'N', matrix \(Q\) is not computed.
- If compq = 'I', \(Q\) is initialized to the unit matrix, and the orthogonal/unitary matrix \(Q\) is returned;
- If compq = 'V', \(Q\) must contain an orthogonal/unitary matrix \(Q_1\) on entry, and the product \(Q_1^* Q\) is returned.

compz

- Must be 'N', 'I', or 'V'.
- If compz = 'N', matrix \(Z\) is not computed.
- If compz = 'I', \(Z\) is initialized to the unit matrix, and the orthogonal/unitary matrix \(Z\) is returned;
If \( \text{compz} = 'V' \), \( Z \) must contain an orthogonal/unitary matrix \( Z_1 \) on entry, and the product \( Z_1^*Z \) is returned.

The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

\( \text{ilo}, \text{ihi} \)

\( \text{ilo} \) and \( \text{ihi} \) mark the rows and columns of \( A \) which are to be reduced. It is assumed that \( A \) is already upper triangular in rows and columns 1:\( \text{ilo} \)-1 and \( \text{ihi}+1:n \). Values of \( \text{ilo} \) and \( \text{ihi} \) are normally set by a previous call to \text{ggbal}; otherwise they should be set to 1 and \( n \) respectively.

Constraint:

If \( n > 0 \), then \( 1 \leq \text{ilo} \leq \text{ihi} \leq n \);
if \( n = 0 \), then \( \text{ilo} = 1 \) and \( \text{ihi} = 0 \).

\( a, b, q, z \)

Arrays:

\( a \) (size \( \max(1, lda*n) \)) contains the \( n \)-by-\( n \) general matrix \( A \).
\( b \) (size \( \max(1, ldb*n) \)) contains the \( n \)-by-\( n \) upper triangular matrix \( B \).
\( q \) (size \( \max(1, ldq*n) \))

If \( \text{compq} = 'N' \), then \( q \) is not referenced.
If \( \text{compq} = 'V' \), then \( q \) must contain the orthogonal/unitary matrix \( Q_1 \), typically from the \( QR \) factorization of \( B \).

\( z \) (size \( \max(1, ldz*n) \))

If \( \text{compz} = 'N' \), then \( z \) is not referenced.
If \( \text{compz} = 'V' \), then \( z \) must contain the orthogonal/unitary matrix \( Z_1 \).

\( lda \)

The leading dimension of \( a \); at least \( \max(1, n) \).

\( ldb \)

The leading dimension of \( b \); at least \( \max(1, n) \).

\( ldq \)

The leading dimension of \( q \);
If \( \text{compq} = 'N' \), then \( ldq \geq 1 \).
If \( \text{compq} = 'I' \) or \( 'V' \), then \( ldq \geq \max(1, n) \).

\( ldz \)

The leading dimension of \( z \);
If \( \text{compz} = 'N' \), then \( ldz \geq 1 \).
If \( \text{compz} = 'I' \) or \( 'V' \), then \( ldz \geq \max(1, n) \).

Output Parameters

\( a \)

On exit, the upper triangle and the first subdiagonal of \( A \) are overwritten with the upper Hessenberg matrix \( H \), and the rest is set to zero.

\( b \)

On exit, overwritten by the upper triangular matrix \( T = \text{Q}^H B Z \). The elements below the diagonal are set to zero.

\( q \)

If \( \text{compq} = 'I' \), then \( q \) contains the orthogonal/unitary matrix \( Q_1 \);
If \( \text{compq} = 'V' \), then \( q \) is overwritten by the product \( Q_1^*Q \).

\( z \)

If \( \text{compz} = 'I' \), then \( z \) contains the orthogonal/unitary matrix \( Z_1 \);
If \( \text{compz} = 'V' \), then \( z \) is overwritten by the product \( Z_1^* Z \).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

**ggbal**

*Balances a pair of general real or complex matrices.*

**Syntax**

```c
lapack_int LAPACKE_sggbal( int matrix_layout, char job, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale );
```

```c
lapack_int LAPACKE_dggbal( int matrix_layout, char job, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale );
```

```c
lapack_int LAPACKE_cggbal( int matrix_layout, char job, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale );
```

```c
lapack_int LAPACKE_zggbal( int matrix_layout, char job, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale );
```

**Include Files**

- mkl.h

**Description**

The routine balances a pair of general real/complex matrices \((A, B)\). This involves, first, permuting \( A \) and \( B \) by similarity transformations to isolate eigenvalues in the first \( 1 \) to \( \text{ilo} - 1 \) and last \( \text{ihi} + 1 \) to \( n \) elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns \( \text{ilo} \) to \( \text{ihi} \) to make the rows and columns as close in norm as possible. Both steps are optional. Balancing may reduce the 1-norm of the matrices, and improve the accuracy of the computed eigenvalues and/or eigenvectors in the generalized eigenvalue problem \( A^* x = \lambda B^* x \).

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **job**
  
  Specifies the operations to be performed on \( A \) and \( B \). Must be 'N' or 'P' or 'S' or 'B'.
  
  If \( \text{job} = 'N' \), then no operations are done; simply set \( \text{ilo} = 1, \text{ihi} = n \), \( \text{lscale}[i] = 1.0 \) and \( \text{rscale}[i] = 1.0 \) for \( i = 0, \ldots, n - 1 \).
  
  If \( \text{job} = 'P' \), then permute only.
  
  If \( \text{job} = 'S' \), then scale only.
If \( \text{job} = \text{'B'} \), then both permute and scale.

The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

Arrays:

- \( a \) (size \( \max(1, \text{lda}^*n) \)) contains the matrix \( A \).
- \( b \) (size \( \max(1, \text{ldb}^*n) \)) contains the matrix \( B \).

If \( \text{job} = \text{'N'} \), \( a \) and \( b \) are not referenced.

The leading dimension of \( a \); at least \( \max(1, n) \).

The leading dimension of \( b \); at least \( \max(1, n) \).

Output Parameters

- \( a, b \) Overwritten by the balanced matrices \( A \) and \( B \), respectively.
- \( i lo, \text{hi} \) are set to integers such that on exit \( A_{i,j} = 0 \) and \( B_{i,j} = 0 \) if \( i > j \) and \( j=1, \ldots, i lo -1 \) or \( i = i hi + 1, \ldots, n \).

If \( \text{job} = \text{'N'} \) or \( \text{'S'} \), then \( i lo = 1 \) and \( i hi = n \).

- \( \text{lscale, rscale} \)

\( \text{lscale} \) contains details of the permutations and scaling factors applied to the left side of \( A \) and \( B \).

If \( P_j \) is the index of the row interchanged with row \( j \), and \( D_j \) is the scaling factor applied to row \( j \), then

\[
\text{lscale}[j - 1] = P_j, \text{ for } j = 1, \ldots, i lo -1
\]
\[
= D_j, \text{ for } j = i lo, \ldots, i hi
\]
\[
= P_j, \text{ for } j = i hi + 1, \ldots, n.
\]

\( \text{rscale} \) contains details of the permutations and scaling factors applied to the right side of \( A \) and \( B \).

If \( P_j \) is the index of the column interchanged with column \( j \), and \( D_j \) is the scaling factor applied to column \( j \), then

\[
\text{rscale}[j - 1] = P_j, \text{ for } j = 1, \ldots, i lo -1
\]
\[
= D_j, \text{ for } j = i lo, \ldots, i hi
\]
\[
= P_j, \text{ for } j = i hi + 1, \ldots, n
\]

The order in which the interchanges are made is \( n \) to \( i hi + 1 \), then \( 1 \) to \( i lo -1 \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

\texttt{?ggbak}

Forms the right or left eigenvectors of a generalized eigenvalue problem.
Syntax

```c
lapack_int LAPACKE_sggbak( int matrix_layout, char job, char side, lapack_int n, 
  lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int m, 
  float* v, lapack_int ldv );

lapack_int LAPACKE_dggbak( int matrix_layout, char job, char side, lapack_int n, 
  lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int m, 
  double* v, lapack_int ldv );

lapack_int LAPACKE_cggbak( int matrix_layout, char job, char side, lapack_int n, 
  lapack_int ilo, lapack_int ihi, const float* lscale, const float* rscale, lapack_int m, 
  lapack_complex_float* v, lapack_int ldv );

lapack_int LAPACKE_zggbak( int matrix_layout, char job, char side, lapack_int n, 
  lapack_int ilo, lapack_int ihi, const double* lscale, const double* rscale, lapack_int m, 
  lapack_complex_double* v, lapack_int ldv );
```

Include Files
- `mkl.h`

Description

The routine forms the right or left eigenvectors of a real/complex generalized eigenvalue problem

\[ A^*x = \lambda^*B^*x \]

by backward transformation on the computed eigenvectors of the balanced pair of matrices output by `ggbal`.

Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **job**
  - Specifies the type of backward transformation required. Must be 'N', 'P', 'S', or 'B'.
    - If `job` = 'N', then no operations are done; return.
    - If `job` = 'P', then do backward transformation for permutation only.
    - If `job` = 'S', then do backward transformation for scaling only.
    - If `job` = 'B', then do backward transformation for both permutation and scaling. This argument must be the same as the argument `job` supplied to `ggbal`.

- **side**
  - Must be 'L' or 'R'.
    - If `side` = 'L', then `v` contains left eigenvectors.
    - If `side` = 'R', then `v` contains right eigenvectors.

- **n**
  - The number of rows of the matrix `V` \( (n \geq 0) \).

- **ilo, ihi**
  - The integers `ilo` and `ihi` determined by `ggbal`. Constraint:
    - If `n > 0`, then \( 1 \leq ilo \leq ihi \leq n \);
    - if `n = 0`, then `ilo = 1` and `ihi = 0`.

- **lscale, rscale**
  - Arrays, size at least `max(1, n)`. 
The array `lscale` contains details of the permutations and/or scaling factors applied to the left side of `A` and `B`, as returned by ?ggbal.

The array `rscale` contains details of the permutations and/or scaling factors applied to the right side of `A` and `B`, as returned by ?ggbal.

The number of columns of the matrix `V` ($m \geq 0$).

Array `v` (size max(1, `ldv`*`m`) for column major layout and max(1, `ldv`*`n`) for row major layout). Contains the matrix of right or left eigenvectors to be transformed, as returned by tgevc.

The leading dimension of `v`; at least max(1, `n`) for column major layout and at least max(1, `m`) for row major layout.

Output Parameters

`v` Overwritten by the transformed eigenvectors

Return Values

This function returns a value `info`.

If `info`=0, the execution is successful.

If `info` = `-i`, the `i`-th parameter had an illegal value.

?gghd3

Reduces a pair of matrices to generalized upper Hessenberg form.

Syntax

```c
lapack_int LAPACKE_sgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, float * a, lapack_int lda, float * b, lapack_int ldb,
float * q, lapack_int ldq, float * z, lapack_int ldz);

lapack_int LAPACKE_dgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, double * a, lapack_int lda, double * b, lapack_int ldb,
double * q, lapack_int ldq, double * z, lapack_int ldz);

lapack_int LAPACKE_cgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, lapack_complex_float * q, lapack_int ldq,
lapack_complex_float * z, lapack_int ldz);

lapack_int LAPACKE_zgghd3 (int matrix_layout, char compq, char compz, lapack_int n,
lapack_int ilo, lapack_int ihi, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, lapack_complex_double * q, lapack_int ldq,
lapack_complex_double * z, lapack_int ldz);
```

Include Files

- mkl.h
Description

?gghd3 reduces a pair of real or complex matrices \((A, B)\) to generalized upper Hessenberg form using orthogonal/unitary transformations, where \(A\) is a general matrix and \(B\) is upper triangular. The form of the generalized eigenvalue problem is

\[ A^*x = \lambda^*B^*x, \]

and \(B\) is typically made upper triangular by computing its QR factorization and moving the orthogonal/unitary matrix \(Q\) to the left side of the equation.

This subroutine simultaneously reduces \(A\) to a Hessenberg matrix \(H\):

\[ Q^T A^* Z = H \] for real flavors

or

\[ Q^T A^* Z = H \] for complex flavors

and transforms \(B\) to another upper triangular matrix \(T\):

\[ Q^T B^* Z = T \] for real flavors

or

\[ Q^T B^* Z = T \] for complex flavors

in order to reduce the problem to its standard form

\[ H^*y = \lambda^*T^*y \]

where \(y = Z^T x\) for real flavors

or

\(y = Z^T x\) for complex flavors.

The orthogonal/unitary matrices \(Q\) and \(Z\) are determined as products of Givens rotations. They may either be formed explicitly, or they may be postmultiplied into input matrices \(Q_1\) and \(Z_1\), so that

for real flavors:

\[ Q_1 \ast A \ast Z_1^T = (Q_1 \ast Q) \ast H \ast (Z_1 \ast Z)^T \]

\[ Q_1 \ast B \ast Z_1^T = (Q_1 \ast Q) \ast T \ast (Z_1 \ast Z)^T \]

for complex flavors:

\[ Q_1 \ast A \ast Z_1^H = (Q_1 \ast Q) \ast H \ast (Z_1 \ast Z)^T \]

\[ Q_1 \ast B \ast Z_1^T = (Q_1 \ast Q) \ast T \ast (Z_1 \ast Z)^T \]

If \(Q_1\) is the orthogonal/unitary matrix from the QR factorization of \(B\) in the original equation \(A^*x = \lambda^*B^*x\), then ?gghd3 reduces the original problem to generalized Hessenberg form.

This is a blocked variant of ?gghrd, using matrix-matrix multiplications for parts of the computation to enhance performance.

Input Parameters

\(matrix\_layout\)

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\(compq\)

\(= 'N':\) do not compute \(Q\);

\(= 'I':\) \(Q\) is initialized to the unit matrix, and the orthogonal/unitary matrix \(Q\) is returned;
= 'V': q must contain an orthogonal/unitary matrix $Q_1$ on entry, and the product $Q_1^*q$ is returned.

$compz$  
= 'N': do not compute $z$;  
= 'I': $z$ is initialized to the unit matrix, and the orthogonal/unitary matrix $Z$ is returned;  
= 'V': $z$ must contain an orthogonal/unitary matrix $Z_1$ on entry, and the product $Z_1^*z$ is returned.

$n$  
The order of the matrices $A$ and $B$.  
$n \geq 0$.

$ilo, ihi$  
$ilo$ and $ihi$ mark the rows and columns of $a$ which are to be reduced. It is assumed that $a$ is already upper triangular in rows and columns $1:ilo - 1$ and $ihi + 1:n$. $ilo$ and $ihi$ are normally set by a previous call to $?ggbal$; otherwise they should be set to 1 and $n$, respectively.  
$1 \leq ilo \leq ihi \leq n$, if $n > 0$; $ilo = 1$ and $ihi = 0$, if $n = 0$.

$a$  
Array, size ($lda \times n$).  
On entry, the $n$-by-$n$ general matrix to be reduced.

$lda$  
The leading dimension of the array $a$.  
$lda \geq \max(1,n)$.

$b$  
Array, ($ldb \times n$).  
On entry, the $n$-by-$n$ upper triangular matrix $B$.

$ldb$  
The leading dimension of the array $b$.  
$ldb \geq \max(1,n)$.

$q$  
Array, size ($ldq \times n$).  
On entry, if $compq = 'V$', the orthogonal/unitary matrix $Q_1$, typically from the QR factorization of $b$.

$ldq$  
The leading dimension of the array $q$.  
$ldq \geq n$ if $compq = 'V'$ or 'I'; $ldq \geq 1$ otherwise.

$z$  
Array, size ($ldz \times n$).  
On entry, if $compz = 'V$', the orthogonal/unitary matrix $Z_1$.  
Not referenced if $compz = 'N$'.

$ldz$  
The leading dimension of the array $z$.  $ldz \geq n$ if $compz = 'V'$ or 'I'; $ldz \geq 1$ otherwise.

**Output Parameters**

$a$  
On exit, the upper triangle and the first subdiagonal of $a$ are overwritten with the upper Hessenberg matrix $H$, and the rest is set to zero.
On exit, the upper triangular matrix $T = Q^T B Z$ for real flavors or $T = Q^H B Z$ for complex flavors. The elements below the diagonal are set to zero.

On exit, if $\text{compq} = 'I'$, the orthogonal/unitary matrix $Q$, and if $\text{compq} = 'V'$, the product $Q_1^* Q$.
Not referenced if $\text{compq} = 'N'$.

On exit, if $\text{compz} = 'I'$, the orthogonal/unitary matrix $Z$, and if $\text{compz} = 'V'$, the product $Z_1^* Z$.
Not referenced if $\text{compz} = 'N'$.

Return Values
This function returns a value $\text{info}$.

= 0: successful exit.
< 0: if $\text{info} = -i$, the $i$-th argument had an illegal value.

Application Notes
This routine reduces $A$ to Hessenberg form and maintains $B$ in using a blocked variant of Moler and Stewart's original algorithm, as described by Kagstrom, Kressner, Quintana-Orti, and Quintana-Orti (BIT 2008).

$hgeqz$
Implements the QZ method for finding the generalized eigenvalues of the matrix pair $(H, T)$.

Syntax

```c
lapack_int LAPACKE_shgeqz( int matrix_layout, char job, char compq, char compz,
                        lapack_int n, lapack_int ilo, lapack_int ihi, float* h, lapack_int ldh, float* t,
                        lapack_int ldt, float* alphar, float* alphai, float* beta, float* q, lapack_int ldq,
                        float* z, lapack_int ldz );

lapack_int LAPACKE_dhgeqz( int matrix_layout, char job, char compq, char compz,
                        lapack_int n, lapack_int ilo, lapack_int ihi, double* h, lapack_int ldh, double* t,
                        lapack_int ldt, double* alphar, double* alphai, double* beta, double* q, lapack_int ldq,
                        double* z, lapack_int ldz );

lapack_int LAPACKE_chgeqz( int matrix_layout, char job, char compq, char compz,
                        lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_float* h, lapack_int ldh,
                        lapack_complex_float* t, lapack_int ldt, lapack_complex_float* alpha,
                        lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
                        lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhgeqz( int matrix_layout, char job, char compq, char compz,
                        lapack_int n, lapack_int ilo, lapack_int ihi, lapack_complex_double* h, lapack_int ldh,
                        lapack_complex_double* t, lapack_int ldt, lapack_complex_double* alpha,
                        lapack_complex_double* beta, lapack_complex_double* q, lapack_int ldq,
                        lapack_complex_double* z, lapack_int ldz );
```

Include Files
- mkl.h
Description

The routine computes the eigenvalues of a real/complex matrix pair \((H, T)\), where \(H\) is an upper Hessenberg matrix and \(T\) is upper triangular, using the double-shift version (for real flavors) or single-shift version (for complex flavors) of the QZ method. Matrix pairs of this type are produced by the reduction to generalized upper Hessenberg form of a real/complex matrix pair \((A, B)\):

\[
A = Q_1^* H Z_1^H, \quad B = Q_1^* T Z_1^H,
\]

as computed by ?gghrd.

For real flavors:

If \(job = 'S'\), then the Hessenberg-triangular pair \((H, T)\) is reduced to generalized Schur form,

\[
H = Q^* S Z^T, \quad T = Q^* P Z^T,
\]

where \(Q\) and \(Z\) are orthogonal matrices, \(P\) is an upper triangular matrix, and \(S\) is a quasi-triangular matrix with 1-by-1 and 2-by-2 diagonal blocks. The 1-by-1 blocks correspond to real eigenvalues of the matrix pair \((H, T)\) and the 2-by-2 blocks correspond to complex conjugate pairs of eigenvalues.

Additionally, the 2-by-2 upper triangular diagonal blocks of \(P\) corresponding to 2-by-2 blocks of \(S\) are reduced to positive diagonal form, that is, if \(S_{j+1,j}\) is non-zero, then \(P_{j+1,j} = P_{j,j+1} = 0\), \(P_{j,j} > 0\), and \(P_{j+1,j+1} > 0\).

For complex flavors:

If \(job = 'S'\), then the Hessenberg-triangular pair \((H, T)\) is reduced to generalized Schur form,

\[
H = Q^* S Z_H^T, \quad T = Q^* P Z_H^T,
\]

where \(Q\) and \(Z\) are unitary matrices, and \(S\) and \(P\) are upper triangular.

For all function flavors:

Optionally, the orthogonal/unitary matrix \(Q\) from the generalized Schur factorization may be post-multiplied by an input matrix \(Q_1\), and the orthogonal/unitary matrix \(Z\) may be post-multiplied by an input matrix \(Z_1\).

If \(Q_1\) and \(Z_1\) are the orthogonal/unitary matrices from ?gghrd that reduced the matrix pair \((A, B)\) to generalized upper Hessenberg form, then the output matrices \(Q_1 Q\) and \(Z_1 Z\) are the orthogonal/unitary factors from the generalized Schur factorization of \((A, B)\):

\[
A = (Q_1 Q)^* S^* (Z_1 Z)^H, \quad B = (Q_1 Q)^* P^* (Z_1 Z)^H.
\]

To avoid overflow, eigenvalues of the matrix pair \((H, T)\) (equivalently, of \((A, B)\)) are computed as a pair of values \((\alpha, \beta)\). For chgeqz/zhgeqz, \(\alpha\) and \(\beta\) are complex, and for shgeqz/dhgeqz, \(\alpha\) is complex and \(\beta\) real. If \(\beta\) is nonzero, \(\lambda = \alpha/\beta\) is an eigenvalue of the generalized nonsymmetric eigenvalue problem (GNEP)

\[
A^* x = \lambda B^* x
\]

and if \(\alpha\) is nonzero, \(\mu = \beta/\alpha\) is an eigenvalue of the alternate form of the GNEP

\[
\mu A^* y = B^* y.
\]

Real eigenvalues (for real flavors) or the values of \(\alpha\) and \(\beta\) for the \(i\)-th eigenvalue (for complex flavors) can be read directly from the generalized Schur form:

\[
\alpha_i = S_{i,i}, \quad \beta_i = P_{i,i}.
\]

Input Parameters

\[
\text{matrix\_layout}
\]

Specifies whether matrix storage layout is row major \((\text{LAPACK\_ROW\_MAJOR})\) or column major \((\text{LAPACK\_COL\_MAJOR})\).

\[
\text{job}
\]

Specifies the operations to be performed. Must be \('E'\) or \('S'\).
If $job = 'E'$, then compute eigenvalues only;
If $job = 'S'$, then compute eigenvalues and the Schur form.

**compq**
Must be 'N', 'I', or 'V'.
If $compq = 'N'$, left Schur vectors ($q$) are not computed;
If $compq = 'I'$, $q$ is initialized to the unit matrix and the matrix of left
Schur vectors of $(H,T)$ is returned;
If $compq = 'V'$, $q$ must contain an orthogonal/unitary matrix $Q_1$ on entry
and the product $Q_1^*Q$ is returned.

**compz**
Must be 'N', 'I', or 'V'.
If $compz = 'N'$, right Schur vectors ($z$) are not computed;
If $compz = 'I'$, $z$ is initialized to the unit matrix and the matrix of right
Schur vectors of $(H,T)$ is returned;
If $compz = 'V'$, $z$ must contain an orthogonal/unitary matrix $Z_1$ on entry
and the product $Z_1^*Z$ is returned.

**n**
The order of the matrices $H$, $T$, $Q$, and $Z$
$(n \geq 0)$.

**ilo, ihi**
$ilo$ and $ihi$ mark the rows and columns of $H$ which are in Hessenberg form.
It is assumed that $H$ is already upper triangular in rows and columns $1:ilo-1$ and
$ihi+1:n$.
Constraint:
If $n > 0$, then $1 \leq ilo \leq ihi \leq n$;
if $n = 0$, then $ilo = 1$ and $ihi = 0$.

**h, t, q, z**
Arrays:
On entry, $h$ (size max(1, $ldh*n$)) contains the $n$-by-$n$ upper Hessenberg
matrix $H$.
On entry, $t$ (size max(1, $ldt*n$)) contains the $n$-by-$n$ upper triangular
matrix $T$.
$q$ (size max(1, $ldq*n$)) :
On entry, if $compq = 'V'$, this array contains the orthogonal/unitary matrix
$Q_1$ used in the reduction of $(A,B)$ to generalized Hessenberg form.
If $compq = 'N'$, then $q$ is not referenced.
$z$ (size max(1, $ldz*n$)) :
On entry, if $compz = 'V'$, this array contains the orthogonal/unitary matrix
$Z_1$ used in the reduction of $(A,B)$ to generalized Hessenberg form.
If $compz = 'N'$, then $z$ is not referenced.

**ldh**
The leading dimension of $h$; at least max(1, $n$).

**ldt**
The leading dimension of $t$; at least max(1, $n$).

**ldq**
The leading dimension of $q$;
If $\text{compq} = 'N'$, then $\text{ldq} \geq 1$.
If $\text{compq} = 'I'$ or 'V', then $\text{ldq} \geq \max(1, n)$.

$\text{ldz}$

The leading dimension of $z$;
If $\text{compq} = 'N'$, then $\text{ldz} \geq 1$.
If $\text{compq} = 'I'$ or 'V', then $\text{ldz} \geq \max(1, n)$.

**Output Parameters**

$h$

*For real flavors:*
If $\text{job} = 'S'$, then on exit $h$ contains the upper quasi-triangular matrix $S$ from the generalized Schur factorization.
If $\text{job} = 'E'$, then on exit the diagonal blocks of $h$ match those of $S$, but the rest of $h$ is unspecified.

*For complex flavors:*
If $\text{job} = 'S'$, then, on exit, $h$ contains the upper triangular matrix $S$ from the generalized Schur factorization.
If $\text{job} = 'E'$, then on exit the diagonal of $h$ matches that of $S$, but the rest of $h$ is unspecified.

$t$

If $\text{job} = 'S'$, then, on exit, $t$ contains the upper triangular matrix $P$ from the generalized Schur factorization.

*For real flavors:*
2-by-2 diagonal blocks of $P$ corresponding to 2-by-2 blocks of $S$ are reduced to positive diagonal form, that is, if $h(j+1,j)$ is non-zero, then $t(j+1,j) = t(j,j+1) = 0$ and $t(j,j)$ and $t(j+1,j+1)$ will be positive.
If $\text{job} = 'E'$, then on exit the diagonal blocks of $t$ match those of $P$, but the rest of $t$ is unspecified.

*For complex flavors:*
if $\text{job} = 'E'$, then on exit the diagonal of $t$ matches that of $P$, but the rest of $t$ is unspecified.

$\text{alphan, alphei}$

Arrays, size at least $\max(1, n)$. The real and imaginary parts, respectively, of each scalar $\text{alpha}$ defining an eigenvalue of GNEP.

If $\text{alphei}[j - 1]$ is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-th eigenvalues are a complex conjugate pair, with $\text{alphei}[j] = -\text{alphei}[j - 1]$.

$\text{alpha}$

Array, size at least $\max(1, n)$.
The complex scalars $\text{alpha}$ that define the eigenvalues of GNEP. $\text{alphei}[i - 1] = S_i, i$ in the generalized Schur factorization.

$\text{beta}$

Array, size at least $\max(1, n)$.
*For real flavors:*
The scalars $\text{beta}$ that define the eigenvalues of GNEP.
Together, the quantities \( \alpha = (\alpha_j[j - 1], \alpha_i[j - 1]) \) and \( \beta = \beta[j - 1] \) represent the \( j \)-th eigenvalue of the matrix pair \((A, B)\), in one of the forms \( \lambda = \alpha / \beta \) or \( \mu = \beta / \alpha \). Since either \( \lambda \) or \( \mu \) may overflow, they should not, in general, be computed.

For complex flavors:

The real non-negative scalars \( \beta \) that define the eigenvalues of GNEP.

\( \beta[i - 1] = P_{i, j} \) in the generalized Schur factorization. Together, the quantities \( \alpha = \alpha[j - 1] \) and \( \beta = \beta[j - 1] \) represent the \( j \)-th eigenvalue of the matrix pair \((A, B)\), in one of the forms \( \lambda = \alpha / \beta \) or \( \mu = \beta / \alpha \). Since either \( \lambda \) or \( \mu \) may overflow, they should not, in general, be computed.

On exit, if \( \text{compq} = 'I' \), \( q \) is overwritten by the orthogonal/unitary matrix of left Schur vectors of the pair \((H, T)\), and if \( \text{compq} = 'V' \), \( q \) is overwritten by the orthogonal/unitary matrix of left Schur vectors of \((A, B)\).

On exit, if \( \text{compz} = 'I' \), \( z \) is overwritten by the orthogonal/unitary matrix of right Schur vectors of the pair \((H, T)\), and if \( \text{compz} = 'V' \), \( z \) is overwritten by the orthogonal/unitary matrix of right Schur vectors of \((A, B)\).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = 1, \ldots, n \), the QZ iteration did not converge.

\((H, T)\) is not in Schur form, but \( \alpha[i - 1], \alpha[i - 1] \) (for real flavors), \( \alpha[i - 1] \) (for complex flavors), and \( \beta[i - 1], i = \text{info}+1, \ldots, n \) should be correct.

If \( \text{info} = n+1, \ldots, 2n \), the shift calculation failed.

\((H, T)\) is not in Schur form, but \( \alpha[i - 1], \alpha[i - 1] \) (for real flavors), \( \alpha[i - 1] \) (for complex flavors), and \( \beta[i - 1], i = \text{info}-n+1, \ldots, n \) should be correct.

\texttt{tgevc}

Computes some or all of the right and/or left generalized eigenvectors of a pair of upper triangular matrices.

Syntax

\[
\begin{align*}
\text{lapack_int LAPACKE_stgevc (int matrix_layout, char side, char howmny, const} \\
\text{lapack_logical* select, lapack_int n, const float* s, lapack_int lds, const float* p,} \\
\text{lapack_int ldp, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int mm,} \\
\text{lapack_int* m);} \\
\text{lapack_int LAPACKE_dtgevc (int matrix_layout, char side, char howmny, const} \\
\text{lapack_logical* select, lapack_int n, const double* s, lapack_int lds, const double* p,} \\
\text{lapack_int ldp, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int mm,} \\
\text{lapack_int* m);} 
\end{align*}
\]
lapack_int LAPACKE_ctgevc (int matrix_layout, char side, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_float* s, lapack_int lds,
const lapack_complex_float* p, lapack_int ldp, lapack_complex_float* vl, lapack_int ldvl,
lapack_complex_float* vr, lapack_int ldvr, lapack_int mm, lapack_int* m);
lapack_int LAPACKE_ztgevc (int matrix_layout, char side, char howmny, const
lapacklogical* select, lapack_int n, const lapack_complex_double* s, lapack_int lds,
const lapack_complex_double* p, lapack_int ldp, lapack_complex_double* vl, lapack_int ldvl,
lapack_complex_double* vr, lapack_int ldvr, lapack_int mm, lapack_int* m);

Include Files
• mkl.h

Description
The routine computes some or all of the right and/or left eigenvectors of a pair of real/complex matrices
$(S,P)$, where $S$ is quasi-triangular (for real flavors) or upper triangular (for complex flavors) and $P$ is upper triangular.

Matrix pairs of this type are produced by the generalized Schur factorization of a real/complex matrix pair
$(A,B)$:
$$A = Q*Z^H, B = Q*Z^H$$
as computed by ?gghrd plus ?hgeqz.

The right eigenvector $x$ and the left eigenvector $y$ of $(S,P)$ corresponding to an eigenvalue $w$ are defined by:
$$S*x = w*P*y, y^H*S = w*y^H*P$$

The eigenvalues are not input to this routine, but are computed directly from the diagonal blocks or diagonal
elements of $S$ and $P$.

This routine returns the matrices $X$ and/or $Y$ of right and left eigenvectors of $(S,P)$, or the products $Z*X$
and/or $Q*Y$, where $Z$ and $Q$ are input matrices.

If $Q$ and $Z$ are the orthogonal/unitary factors from the generalized Schur factorization of a matrix pair $(A,B)$,
then $Z*X$ and $Q*Y$ are the matrices of right and left eigenvectors of $(A,B)$.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
or column major (LAPACK_COL_MAJOR).

side Must be 'R', 'L', or 'B'.
If side = 'R', compute right eigenvectors only.
If side = 'L', compute left eigenvectors only.
If side = 'B', compute both right and left eigenvectors.

howmny Must be 'A', 'B', or 'S'.
If howmny = 'A', compute all right and/or left eigenvectors.
If howmny = 'B', compute all right and/or left eigenvectors, backtransformed by the matrices in $vr$ and/or $vl$.
If howmny = 'S', compute selected right and/or left eigenvectors, specified by the logical array select.

select
Array, size at least max (1, n).

If howmny = 'S', select specifies the eigenvectors to be computed.

If howmny = 'A' or 'B', select is not referenced.

For real flavors:

If \( w[j] \) is a real eigenvalue, the corresponding real eigenvector is computed if select\([j]\) is 1.

If \( w[j] \) and omega\([j + 1]\) are the real and imaginary parts of a complex eigenvalue, the corresponding complex eigenvector is computed if either select\([j]\) or select\([j + 1]\) is 1, and on exit select\([j]\) is set to 1 and select\([j + 1]\) is set to 0.

For complex flavors:

The eigenvector corresponding to the \( j \)-th eigenvalue is computed if select\([j]\) is 1.

The order of the matrices \( S \) and \( P \) \((n \geq 0)\).

Arrays:

\( s \) (size max(1, \( lds*n \))) contains the matrix \( S \) from a generalized Schur factorization as computed by ?hgeqz. This matrix is upper quasi-triangular for real flavors, and upper triangular for complex flavors.

\( p \) (size max(1, \( ldp*n \))) contains the upper triangular matrix \( P \) from a generalized Schur factorization as computed by ?hgeqz.

For real flavors, 2-by-2 diagonal blocks of \( P \) corresponding to 2-by-2 blocks of \( S \) must be in positive diagonal form.

For complex flavors, \( P \) must have real diagonal elements.

If \( side = 'L' \) or 'B' and howmny = 'B', \( vl \)(size max(1, \( ldvl*mm \))) for column major layout and max(1, \( ldvl*n \)) for row major layout must contain an \( n \)-by-\( n \) matrix \( Q \) (usually the orthogonal/unitary matrix \( Q \) of left Schur vectors returned by ?hgeqz).

If \( side = 'R' \), \( vl \) is not referenced.

If \( side = 'R' \) or 'B' and howmny = 'B', \( vr \)(size max(1, \( ldvr*mm \))) for column major layout and max(1, \( ldvr*n \)) for row major layout must contain an \( n \)-by-\( n \) matrix \( Z \) (usually the orthogonal/unitary matrix \( Z \) of right Schur vectors returned by ?hgeqz).

If \( side = 'L' \), \( vr \) is not referenced.

\( lds \)

The leading dimension of \( s \); at least max(1, n).

\( ldp \)

The leading dimension of \( p \); at least max(1, n).

\( ldvl \)

The leading dimension of \( vl \);

If \( side = 'L' \) or 'B', then \( ldvl \geq n \) for column major layout and \( ldvl \geq \text{max}(1, \text{mm}) \) for row major layout.

If \( side = 'R' \), then \( ldvl \geq 1 \).

\( ldvr \)

The leading dimension of \( vr \);
If \( side = 'R' \) or \( 'B' \), then \( ldvr \geq n \) for column major layout and \( ldvr \geq \max(1, mm) \) for row major layout.

If \( side = 'L' \), then \( ldvr \geq 1 \).

\( mm \)

The number of columns in the arrays \( vl \) and/or \( vr \) (\( mm \geq m \)).

**Output Parameters**

\( vl \)

On exit, if \( side = 'L' \) or \( 'B' \), \( vl \) contains:

- if \( howmny = 'A' \), the matrix \( Y \) of left eigenvectors of \((S,P)\);
- if \( howmny = 'B' \), the matrix \( Q^*Y \);
- if \( howmny = 'S' \), the left eigenvectors of \((S,P)\) specified by \select\, stored consecutively in the columns of \( vl \), in the same order as their eigenvalues.

*For real flavors:*

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

\( vr \)

On exit, if \( side = 'R' \) or \( 'B' \), \( vr \) contains:

- if \( howmny = 'A' \), the matrix \( X \) of right eigenvectors of \((S,P)\);
- if \( howmny = 'B' \), the matrix \( Z^*X \);
- if \( howmny = 'S' \), the right eigenvectors of \((S,P)\) specified by \select\, stored consecutively in the columns of \( vr \), in the same order as their eigenvalues.

*For real flavors:*

A complex eigenvector corresponding to a complex eigenvalue is stored in two consecutive columns, the first holding the real part, and the second the imaginary part.

\( m \)

The number of columns in the arrays \( vl \) and/or \( vr \) actually used to store the eigenvectors.

If \( howmny = 'A' \) or \( 'B' \), \( m \) is set to \( n \).

*For real flavors:*

Each selected real eigenvector occupies one column and each selected complex eigenvector occupies two columns.

*For complex flavors:*

Each selected eigenvector occupies one column.

**Return Values**

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

*For real flavors:*

if \( info = i>0 \), the \( (i:i+1) \) does not have a complex eigenvalue.
Reorders the generalized Schur decomposition of a pair of matrices \((A,B)\) so that one diagonal block of \((A,B)\) moves to another row index.

**Syntax**

```c
lapack_int LAPACKE_stgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* q, lapack_int ldq, float* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);

lapack_int LAPACKE_dtgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* q, lapack_int ldq, double* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);

lapack_int LAPACKE_ctgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* q, lapack_int ldq, lapack_complex_float* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);

lapack_int LAPACKE_ztgexc (int matrix_layout, lapack_logical wantq, lapack_logical wantz, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* q, lapack_int ldq, lapack_complex_double* z, lapack_int ldz, lapack_int* ifst, lapack_int* ilst);
```

**Include Files**

- mkl.h

**Description**

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair \((A,B)\) using an orthogonal/unitary equivalence transformation

\[
(A,B) = Q^* (A,B) Z^H,
\]

so that the diagonal block of \((A, B)\) with row index \(ifst\) is moved to row \(ilst\). Matrix pair \((A, B)\) must be in a generalized real-Schur/Schur canonical form (as returned by gges), that is, \(A\) is block upper triangular with 1-by-1 and 2-by-2 diagonal blocks and \(B\) is upper triangular. Optionally, the matrices \(Q\) and \(Z\) of generalized Schur vectors are updated.

\[
\begin{align*}
Q_{in}^* A_{in}^* Z_{in}^T &= Q_{out}^* A_{out}^* Z_{out}^T, \\
Q_{in}^* B_{in}^* Z_{in}^T &= Q_{out}^* B_{out}^* Z_{out}^T.
\end{align*}
\]

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **wantq, wantz**
  If \(wantq = 1\), update the left transformation matrix \(Q\);
  If \(wantq = 0\), do not update \(Q\);
  If \(wantz = 1\), update the right transformation matrix \(Z\);
  If \(wantz = 0\), do not update \(Z\).

- **n**
  The order of the matrices \(A\) and \(B\) \((n \geq 0)\).

- **a, b, q, z**
  Arrays:
a (size max(1, lda*n)) contains the matrix A.
b (size max(1, ldb*n)) contains the matrix B.
q (size at least 1 if wantq = 0 and at least max(1, ldq*n) if wantq = 1)
If wantq = 0, then q is not referenced.
If wantq = 1, then q must contain the orthogonal/unitary matrix Q.
z (size at least 1 if wantz = 0 and at least max(1, ldz*n) if wantz = 1)
If wantz = 0, then z is not referenced.
If wantz = 1, then z must contain the orthogonal/unitary matrix Z.

lda
The leading dimension of a; at least max(1, n).

ldb
The leading dimension of b; at least max(1, n).

ldq
The leading dimension of q;
If wantq = 0, then ldq ≥ 1.
If wantq = 1, then ldq ≥ max(1, n).

ldz
The leading dimension of z;
If wantz = 0, then ldz ≥ 1.
If wantz = 1, then ldz ≥ max(1, n).

ifst, ilst
Specify the reordering of the diagonal blocks of (A, B). The block with row index ifst is moved to row ilst, by a sequence of swapping between adjacent blocks. Constraint: 1 ≤ ifst, ilst ≤ n.

Output Parameters

a, b, q, z
Overwritten by the updated matrices A, B, Q, and Z respectively.

ifst, ilst
Overwritten for real flavors only.
If ifst pointed to the second row of a 2 by 2 block on entry, it is changed to point to the first row; ilst always points to the first row of the block in its final position (which may differ from its input value by ±1).

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = −i, the i-th parameter had an illegal value.
If info = 1, the transformed matrix pair (A, B) would be too far from generalized Schur form; the problem is ill-conditioned. (A, B) may have been partially reordered, and ilst points to the first row of the current position of the block being moved.

?tgsen
Reorders the generalized Schur decomposition of a pair of matrices (A, B) so that a selected cluster of eigenvalues appears in the leading diagonal blocks of (A, B).
Syntax

```c
lapack_int LAPACKE_stgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, float* a, lapack_int lda,
float* b, lapack_int ldb, float* alphar, float* alphai, float* beta, float* q,
lapack_int ldq, float* z, lapack_int ldz, lapack_int* m, float* pl, float* pr, float* dif);
```

```c
lapack_int LAPACKE_dtgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, float* a, lapack_int lda,
double* b, lapack_int ldb, double* alphar, double* alphai, double* beta, double* q,
lapack_int ldq, double* z, lapack_int ldz, lapack_int* m, double* pl, double* pr, double* dif);
```

```c
lapack_int LAPACKE_ctgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, lapack_complex_float* a,
lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* q, lapack_int ldq,
lapack_complex_float* z, lapack_int ldz, lapack_int* m, double* pl, double* pr, double* dif);
```

```c
lapack_int LAPACKE_ztgsen( int matrix_layout, lapack_int ijob, lapack_logical wantq,
lapack_logical wantz, const lapack_logical* select, lapack_int n, lapack_complex_double* a,
lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* alpha,
lapack_complex_double* beta, lapack_complex_double* q, lapack_int ldq,
lapack_complex_double* z, lapack_int ldz, lapack_int* m, double* pl, double* pr, double* dif);
```

Include Files
- mkl.h

Description

The routine reorders the generalized real-Schur/Schur decomposition of a real/complex matrix pair \((A, B)\) (in terms of an orthogonal/unitary equivalence transformation \(Q^* (A, B) \cdot Z\) for real flavors or \(Q^* (A, B) \cdot Z\) for complex flavors), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the pair \((A, B)\). The leading columns of \(Q\) and \(Z\) form orthonormal/unitary bases of the corresponding left and right eigenspaces (deflating subspaces).

\((A, B)\) must be in generalized real-Schur/Schur canonical form (as returned by `gges`), that is, \(A\) and \(B\) are both upper triangular.

`tgsen` also computes the generalized eigenvalues

\[ \omega_j = (\text{alphar}(j) + \text{alphai}(j) \cdot i)/\text{beta}(j) \] (for real flavors)

\[ \omega_j = \text{alpha}(j)/\text{beta}(j) \] (for complex flavors)

of the reordered matrix pair \((A, B)\).

Optionally, the routine computes the estimates of reciprocal condition numbers for eigenvalues and eigenspaces. These are \(\text{Difu}[A_{11}, B_{11}], (A_{22}, B_{22})\) and \(\text{Difl}[A_{11}, B_{11}], (A_{22}, B_{22})\), that is, the separation(s) between the matrix pairs \((A_{11}, B_{11})\) and \((A_{22}, B_{22})\) that correspond to the selected cluster and the eigenvalues outside the cluster, respectively, and norms of "projections" onto left and right eigenspaces with respect to the selected cluster in the \((1,1)\)-block.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**ijob**
Specifies whether condition numbers are required for the cluster of eigenvalues (pl and pr) or the deflating subspaces Difu and Difl.
- If ijob = 0, only reorder with respect to select;
- If ijob = 1, reciprocal of norms of "projections" onto left and right eigenspaces with respect to the selected cluster (pl and pr);
- If ijob = 2, compute upper bounds on Difu and Difl, using F-norm-based estimate (dif (1:2));
- If ijob = 3, compute estimate of Difu and Difl, using 1-norm-based estimate (dif (1:2)). This option is about 5 times as expensive as ijob = 2;
- If ijob = 4, compute pl, pr and dif (i.e., options 0, 1 and 2 above). This is an economic version to get it all;
- If ijob = 5, compute pl, pr and dif (i.e., options 0, 1 and 3 above).

**wantq, wantz**
- If wantq = 1, update the left transformation matrix Q;
- If wantq = 0, do not update Q;
- If wantz = 1, update the right transformation matrix Z;
- If wantz = 0, do not update Z.

**select**
Array, size at least max (1, n). Specifies the eigenvalues in the selected cluster.
- To select an eigenvalue \( \omega_j \), select[j-1] must be 1.
- For real flavors: to select a complex conjugate pair of eigenvalues \( \omega_j \) and \( \omega_j + 1 \) (corresponding 2 by 2 diagonal block), select[j-1] and/or select[j] must be set to 1; the complex conjugate \( \omega_j \) and \( \omega_j + 1 \) must be either both included in the cluster or both excluded.

**n**
The order of the matrices A and B (\( n \geq 0 \)).

**a, b, q, z**
Arrays:
- a (size max(1, lda*n)) contains the matrix A.
  - For real flavors: A is upper quasi-triangular, with (A, B) in generalized real Schur canonical form.
  - For complex flavors: A is upper triangular, in generalized Schur canonical form.
- b (size max(1, ldb*n)) contains the matrix B.
  - For real flavors: B is upper triangular, with (A, B) in generalized real Schur canonical form.
  - For complex flavors: B is upper triangular, in generalized Schur canonical form.
- q (size at least 1 if wantq = 0 and at least max(1, ldq*n) if wantq = 1)
  - If wantq = 1, then q is an \( n \)-by-\( n \) matrix;
If $wantq = 0$, then $q$ is not referenced.

$z$ (size at least 1 if $wantz = 0$ and at least $\max(1, ldz \times n)$ if $wantz = 1$)

If $wantz = 1$, then $z$ is an $n$-by-$n$ matrix;
If $wantz = 0$, then $z$ is not referenced.

$lda$
The leading dimension of $a$; at least $\max(1, n)$.

$ldb$
The leading dimension of $b$; at least $\max(1, n)$.

$ldq$
The leading dimension of $q$; $ldq \geq 1$.

If $wantq = 1$, then $ldq \geq \max(1, n)$.

$ldz$
The leading dimension of $z$; $ldz \geq 1$.

If $wantz = 1$, then $ldz \geq \max(1, n)$.

Output Parameters

$a, b$
Overwritten by the reordered matrices $A$ and $B$, respectively.

$\text{alphan, alphi}$
Arrays, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in real flavors.
See $beta$.

$\text{alpha}$
Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors.
See $beta$.

$beta$
Array, size at least $\max(1, n)$.

For real flavors:
On exit, $(\text{alphan}[j] + \text{alphi}[j] \times i) / \text{beta}[j], j=0,\ldots, n-1$, will be the generalized eigenvalues.

$\text{alphan}[j] + \text{alphi}[j] \times i$ and $\text{beta}[j], j=0,\ldots, n-1$ are the diagonals of the complex Schur form $(S,T)$ that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of $(A,B)$ were further reduced to triangular form using complex unitary transformations.

If $\text{alphi}[j-1]$ is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alphi}[j]$ negative.

For complex flavors:
The diagonal elements of $A$ and $B$, respectively, when the pair $(A,B)$ has been reduced to generalized Schur form. $\text{alpha}[i] / \text{beta}[i], i=0,\ldots, n-1$ are the generalized eigenvalues.

$q$
If $wantq = 1$, then, on exit, $Q$ has been postmultiplied by the left orthogonal transformation matrix which reorder $(A, B)$. The leading $m$ columns of $Q$ form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).
If `wantz = 1`, then, on exit, \( Z \) has been postmultiplied by the left orthogonal transformation matrix which reorder \((A, B)\). The leading \( m \) columns of \( Z \) form orthonormal bases for the specified pair of left eigenspaces (deflating subspaces).

\( m \)

The dimension of the specified pair of left and right eigen-spaces (deflating subspaces); \( 0 \leq m \leq n \).

\( pl, pr \)

If \( ijob = 1, 4, \) or \( 5, \) \( pl \) and \( pr \) are lower bounds on the reciprocal of the norm of "projections" onto left and right eigenspaces with respect to the selected cluster.

\( 0 < pl, pr \leq 1 \). If \( m = 0 \) or \( m = n, pl = pr = 1 \).

If \( ijob = 0, 2 \) or \( 3, \) \( pl \) and \( pr \) are not referenced

\( dif \)

Array, size (2).

If \( ijob \geq 2, \) \( dif(1:2) \) store the estimates of \( Difu \) and \( Difl \).

If \( ijob = 2 \) or \( 4, \) \( dif(1:2) \) are F-norm-based upper bounds on \( Difu \) and \( Difl \).

If \( ijob = 3 \) or \( 5, \) \( dif(1:2) \) are 1-norm-based estimates of \( Difu \) and \( Difl \).

If \( m = 0 \) or \( m = n, dif(1:2) = F\text{-}\text{norm}([A, B]). \)

If \( ijob = 0 \) or \( 1, \) \( dif \) is not referenced

**Return Values**

This function returns a value \( info \).

If \( info=0, \) the execution is successful.

If \( info = -i, \) the \( i \)-th parameter had an illegal value.

If \( info = 1, \) Reordering of \((A, B)\) failed because the transformed matrix pair \((A, B)\) would be too far from generalized Schur form; the problem is very ill-conditioned. \((A, B)\) may have been partially reordered.

If \( ijob > 0, 0 \) is returned in \( dif, pl \) and \( pr \).

**tgsyl**

Solves the generalized Sylvester equation.

**Syntax**

```c
lapack_int LAPACKE_stgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int m, lapack_int n, const float* a, lapack_int lda, const float* b, lapack_int ldb, float* c, lapack_int ldc, const float* d, lapack_int ldd, const float* e, lapack_int lde, float* f, lapack_int ldf, float* scale, float* dif );

lapack_int LAPACKE_dtgsyl( int matrix_layout, char trans, lapack_int ijob, lapack_int m, lapack_int n, const double* a, lapack_int lda, const double* b, lapack_int ldb, double* c, lapack_int ldc, const double* d, lapack_int ldd, const double* e, lapack_int lde, double* f, lapack_int ldf, double* scale, double* dif );
```
Include Files

- mkl.h

Description

The routine solves the generalized Sylvester equation:

\[ A \cdot R - L \cdot B = \text{scale} \cdot C \]
\[ D \cdot R - L \cdot E = \text{scale} \cdot F \]

where \( R \) and \( L \) are unknown \( m \)-by-\( n \) matrices, \((A, D), (B, E)\) and \((C, F)\) are given matrix pairs of size \( m \)-by-\( m \), \( n \)-by-\( n \) and \( m \)-by-\( n \), respectively, with real/complex entries. \((A, D)\) and \((B, E)\) must be in generalized real-Schur/Schur canonical form, that is, \( A, B \) are upper quasi-triangular/triangular and \( D, E \) are upper triangular.

The solution \((R, L)\) overwrites \((C, F)\). The factor \( \text{scale} \), \( 0 \leq \text{scale} \leq 1 \), is an output scaling factor chosen to avoid overflow.

In matrix notation the above equation is equivalent to the following: solve \( Z^T x = \text{scale} \cdot b \), where \( Z \) is defined as

\[
Z = \begin{pmatrix}
\text{kron}(I_k, A) & -\text{kron}(B^T, I_k) \\
\text{kron}(I_k, D) & -\text{kron}(E^T, I_k)
\end{pmatrix}
\]

Here \( I_k \) is the identity matrix of size \( k \) and \( X^T \) is the transpose/conjugate-transpose of \( X \). \( \text{kron}(X, Y) \) is the Kronecker product between the matrices \( X \) and \( Y \).

If \( \text{trans} = 'T' \) (for real flavors), or \( \text{trans} = 'C' \) (for complex flavors), the routine \( ?\text{tgsyl} \) solves the transposed/conjugate-transposed system \( Z^T y = \text{scale} \cdot b \), which is equivalent to solve for \( R \) and \( L \) in

\[ A^T \cdot R + D^T \cdot L = \text{scale} \cdot C \]
\[ R \cdot B^T + L \cdot E^T = \text{scale} \cdot (-F) \]

This case (\( \text{trans} = 'T' \) for \( \text{stgsyl/dtgsyl} \) or \( \text{trans} = 'C' \) for \( \text{ctgsyl/ztgsyl} \)) is used to compute an one-norm-based estimate of \( \text{Dif}[(A, D), (B, E)] \), the separation between the matrix pairs \((A, D)\) and \((B, E)\).

If \( \text{ijob} \geq 1 \), \( ?\text{tgsyl} \) computes a Frobenius norm-based estimate of \( \text{Dif}[(A, D), (B, E)] \). That is, the reciprocal of a lower bound on the reciprocal of the smallest singular value of \( Z \). This is a level 3 BLAS algorithm.

Input Parameters

- \text{matrix_layout} \quad \text{Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).}
**trans**

Must be 'N', 'T', or 'C'.

If \( \text{trans} = \text{'N'} \), solve the generalized Sylvester equation.

If \( \text{trans} = \text{'T'} \), solve the 'transposed' system (for real flavors only).

If \( \text{trans} = \text{'C'} \), solve the 'conjugate transposed' system (for complex flavors only).

**ijob**

Specifies what kind of functionality to be performed:

If \( \text{ijob} = 0 \), solve the generalized Sylvester equation only;

If \( \text{ijob} = 1 \), perform the functionality of \( \text{ijob} = 0 \) and \( \text{ijob} = 3 \);

If \( \text{ijob} = 2 \), perform the functionality of \( \text{ijob} = 0 \) and \( \text{ijob} = 4 \);

If \( \text{ijob} = 3 \), only an estimate of \( \text{Dif}([A, D], [B, E]) \) is computed (look ahead strategy is used);

If \( \text{ijob} = 4 \), only an estimate of \( \text{Dif}([A, D], [B, E]) \) is computed (?gecon on sub-systems is used). If \( \text{trans} = \text{'T'} \) or \( \text{'C'} \), \( \text{ijob} \) is not referenced.

**m**

The order of the matrices \( A \) and \( D \), and the row dimension of the matrices \( C, F, R \) and \( L \).

**n**

The order of the matrices \( B \) and \( E \), and the column dimension of the matrices \( C, F, R \) and \( L \).

**a, b, c, d, e, f**

Arrays:

- \( a \) (size \( \text{max}(1, \text{lda} \times m) \)) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix \( A \).
- \( b \) (size \( \text{max}(1, \text{ldb} \times n) \)) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix \( B \).
- \( c \) (size \( \text{max}(1, \text{ldc} \times n) \) for column major layout and \( \text{max}(1, \text{ldc} \times m) \) for row major layout) contains the right-hand-side of the first matrix equation in the generalized Sylvester equation (as defined by \( \text{trans} \)).
- \( d \) (size \( \text{max}(1, \text{ldd} \times m) \)) contains the upper triangular matrix \( D \).
- \( e \) (size \( \text{max}(1, \text{lde} \times n) \)) contains the upper triangular matrix \( E \).
- \( f \) (size \( \text{max}(1, \text{ldf} \times n) \) for column major layout and \( \text{max}(1, \text{ldf} \times m) \) for row major layout) contains the right-hand-side of the second matrix equation in the generalized Sylvester equation (as defined by \( \text{trans} \)).

**lda**

The leading dimension of \( a \); at least \( \text{max}(1, m) \).

**ldb**

The leading dimension of \( b \); at least \( \text{max}(1, n) \).

**ldc**

The leading dimension of \( c \); at least \( \text{max}(1, m) \) for column major layout and at least \( \text{max}(1, n) \) for row major layout.

**ldd**

The leading dimension of \( d \); at least \( \text{max}(1, m) \).

**lde**

The leading dimension of \( e \); at least \( \text{max}(1, n) \).

**ldf**

The leading dimension of \( f \); at least \( \text{max}(1, m) \) for column major layout and at least \( \text{max}(1, n) \) for row major layout.
Output Parameters

\( c \)

If \( ijob=0, 1, \) or \( 2, \) overwritten by the solution \( R. \)

If \( ijob=3 \) or \( 4 \) and \( trans = 'N', \) \( c \) holds \( R, \) the solution achieved during the computation of the Dif-estimate.

\( f \)

If \( ijob=0, 1, \) or \( 2, \) overwritten by the solution \( L. \)

If \( ijob=3 \) or \( 4 \) and \( trans = 'N', \) \( f \) holds \( L, \) the solution achieved during the computation of the Dif-estimate.

\( dif \)

On exit, \( dif \) is the reciprocal of a lower bound of the reciprocal of the Dif-function, that is, \( dif \) is an upper bound of \( \text{Dif}[(A, D), (B, E)] = \sigma_{\text{min}}(Z), \) where \( Z \) as defined in the description.

If \( ijob = 0, \) or \( trans = 'T' \) (for real flavors), or \( trans = 'C' \) (for complex flavors), \( dif \) is not touched.

\( scale \)

On exit, \( scale \) is the scaling factor in the generalized Sylvester equation.

If \( 0 < scale < 1, \) \( c \) and \( f \) hold the solutions \( R \) and \( L, \) respectively, to a slightly perturbed system but the input matrices \( A, B, D \) and \( E \) have not been changed.

If \( scale = 0, \) \( c \) and \( f \) hold the solutions \( R \) and \( L, \) respectively, to the homogeneous system with \( C = F = 0. \) Normally, \( scale = 1. \)

Return Values

This function returns a value \( info. \)

If \( info=0, \) the execution is successful.

If \( info = -i, \) the \( i-th \) parameter had an illegal value.

If \( info > 0, (A, D) \) and \( (B, E) \) have common or close eigenvalues.

?tgsna

Estimates reciprocal condition numbers for specified eigenvalues and/or eigenvectors of a pair of matrices in generalized real Schur canonical form.

Syntax

\[
\text{lapack_int LAPACKE_stgsna}(\text{int} \text{ matrix_layout}, \text{char} \text{ job}, \text{char} \text{ howmny}, \text{const lapack_logical* select}, \text{lapack_int n}, \text{const float* a, lapack_int lda}, \text{const float* b, lapack_int ldb, const float* vl, lapack_int ldvl, const float* vr, lapack_int ldvr, float* s, float* dif, lapack_int mm, lapack_int* m});
\]

\[
\text{lapack_int LAPACKE_dtgsna}(\text{int} \text{ matrix_layout}, \text{char} \text{ job}, \text{char} \text{ howmny}, \text{const lapack_logical* select}, \text{lapack_int n}, \text{const double* a, lapack_int lda}, \text{const double* b, lapack_int ldb, const double* vl, lapack_int ldvl, const double* vr, lapack_int ldvr, double* s, double* dif, lapack_int mm, lapack_int* m});
\]

\[
\text{lapack_int LAPACKE_ctgsna}(\text{int} \text{ matrix_layout}, \text{char} \text{ job}, \text{char} \text{ howmny}, \text{const lapack_logical* select}, \text{lapack_int n}, \text{const lapack_complex_float* a, lapack_int lda}, \text{const lapack_complex_float* b, lapack_int ldb, const lapack_complex_float* vl, lapack_int ldvl, const lapack_complex_float* vr, lapack_int ldvr, float* s, float* dif, lapack_int mm, lapack_int* m});
\]
lapack_int LAPACKE_ztgsna( int matrix_layout, char job, char howmny, const
lapack_logical* select, lapack_int n, const lapack_complex_double* a, lapack_int lda,
const lapack_complex_double* b, lapack_int ldb, const lapack_complex_double* vl,
lapack_int ldvl, const lapack_complex_double* vr, lapack_int ldvr, double* s, double* dif, lapack_int mm,
lapack_int* m );

Include Files
• mkl.h

Description
The real flavors stgsna/dtgsna of this routine estimate reciprocal condition numbers for specified
eigenvalues and/or eigenvectors of a matrix pair (A, B) in generalized real Schur canonical form (or of any
matrix pair (Q*A*Z^T, Q*B*Z^T) with orthogonal matrices Q and Z.

(A, B) must be in generalized real Schur form (as returned by gges/gges), that is, A is block upper triangular
with 1-by-1 and 2-by-2 diagonal blocks. B is upper triangular.

The complex flavors ctgsna/ztgsna estimate reciprocal condition numbers for specified eigenvalues and/or
eigenvectors of a matrix pair (A, B). (A, B) must be in generalized Schur canonical form, that is, A and B are
both upper triangular.

Input Parameters

matrix_layout specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
or column major (LAPACK_COL_MAJOR).

job specifies whether condition numbers are required for eigenvalues or
eigenvectors. Must be 'E' or 'V' or 'B'.

If job = 'E', for eigenvalues only (compute s ).
If job = 'V', for eigenvectors only (compute dif ).
If job = 'B', for both eigenvalues and eigenvectors (compute both s and
dif ).

howmny must be 'A' or 'S'.
If howmny = 'A', compute condition numbers for all eigenpairs.
If howmny = 'S', compute condition numbers for selected eigenpairs
specified by the logical array select.

select array, size at least max (1, n).
If howmny = 'S', select specifies the eigenpairs for which condition
numbers are required.
If howmny = 'A', select is not referenced.

For real flavors:
To select condition numbers for the eigenpair corresponding to a real
eigenvalue \( \omega_j \), select[j - 1] must be set to 1; to select condition numbers corresponding to a complex conjugate pair of eigenvalues \( \omega_j \) and \( \omega_j + 1 \),
either select[j - 1] or select[j] must be set to 1.

For complex flavors:
To select condition numbers for the corresponding \( j \)-th eigenvalue and/or eigenvector, \( \text{select}[j - 1] \) must be set to 1.

The order of the square matrix pair \((A, B)\) \((n \geq 0)\).

Arrays:

- \( a \) (size \( \max(1, lda*\nu) \)) contains the upper quasi-triangular (for real flavors) or upper triangular (for complex flavors) matrix \( A \) in the pair \((A, B)\).
- \( b \) (size \( \max(1, ldb*\nu) \)) contains the upper triangular matrix \( B \) in the pair \((A, B)\).

If \( \text{job} = 'E' \) or \( 'B' \), \( vl \) (size \( \max(1, ldvl*\mu) \)) for column major layout and \( \max(1, ldvl*\nu) \) for row major layout) must contain left eigenvectors of \((A, B)\), corresponding to the eigenpairs specified by \( \text{howmny} \) and \( \text{select} \). The eigenvectors must be stored in consecutive columns of \( vl \), as returned by \(? \text{tgevc}\).

If \( \text{job} = 'V' \), \( vl \) is not referenced.

If \( \text{job} = 'E' \) or \( 'B' \), \( vr \) (size \( \max(1, ldvr*\mu) \)) for column major layout and \( \max(1, ldvr*\nu) \) for row major layout) must contain right eigenvectors of \((A, B)\), corresponding to the eigenpairs specified by \( \text{howmny} \) and \( \text{select} \). The eigenvectors must be stored in consecutive columns of \( vr \), as returned by \(? \text{tgevc}\).

If \( \text{job} = 'V' \), \( vr \) is not referenced.

- \( lda \) The leading dimension of \( a \); at least \( \max(1, n) \).
- \( ldb \) The leading dimension of \( b \); at least \( \max(1, n) \).
- \( ldvl \) The leading dimension of \( vl \); \( ldvl \geq 1 \).
- \( ldvr \) The leading dimension of \( vr \); \( ldvr \geq 1 \).

The number of elements in the arrays \( s \) and \( \text{dif} (mm \geq m) \).

Output Parameters

- \( s \) Array, size \( mm \).

  If \( \text{job} = 'E' \) or \( 'B' \), contains the reciprocal condition numbers of the selected eigenvalues, stored in consecutive elements of the array.

  If \( \text{job} = 'V' \), \( s \) is not referenced.

  For real flavors:

  For a complex conjugate pair of eigenvalues two consecutive elements of \( s \) are set to the same value. Thus, \( s[j - 1] \), \( \text{dif}[j - 1] \), and the \( j \)-th columns of \( vl \) and \( vr \) all correspond to the same eigenpair (but not in general the \( j \)-th eigenpair, unless all eigenpairs are selected).
$dif$

Array, size $mm$.

If $job = 'V' \text{ or } 'B'$, contains the estimated reciprocal condition numbers of the selected eigenvectors, stored in consecutive elements of the array.

If the eigenvalues cannot be reordered to compute $dif[j]$, $dif[j]$ is set to 0; this can only occur when the true value would be very small anyway.

If $job = 'E'$, $dif$ is not referenced.

For real flavors:

For a complex eigenvector, two consecutive elements of $dif$ are set to the same value.

For complex flavors:

For each eigenvalue/vector specified by $select$, $dif$ stores a Frobenius norm-based estimate of $Difl$.

$m$

The number of elements in the arrays $s$ and $dif$ used to store the specified condition numbers; for each selected eigenvalue one element is used.

If $howmny = 'A'$, $m$ is set to $n$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

Generalized Singular Value Decomposition: LAPACK Computational Routines

This section describes LAPACK computational routines used for finding the generalized singular value decomposition (GSVD) of two matrices $A$ and $B$ as

\[ U^H A Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, \]
\[ V^H B Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix}, \]

where $U$, $V$, and $Q$ are orthogonal/unitary matrices, $R$ is a nonsingular upper triangular matrix, and $D_1$, $D_2$ are “diagonal” matrices of the structure detailed in the routines description section.

Table “Computational Routines for Generalized Singular Value Decomposition” lists LAPACK routines that perform generalized singular value decomposition of matrices.

**Computational Routines for Generalized Singular Value Decomposition**

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ggsvp</td>
<td>Computes the preprocessing decomposition for the generalized SVD</td>
</tr>
<tr>
<td>ggsvp3</td>
<td>Performs preprocessing for a generalized SVD.</td>
</tr>
<tr>
<td>ggsvd3</td>
<td>Computes generalized SVD.</td>
</tr>
<tr>
<td>tgsja</td>
<td>Computes the generalized SVD of two upper triangular or trapezoidal matrices</td>
</tr>
</tbody>
</table>

You can use routines listed in the above table as well as the driver routine $ggsvd$ to find the GSVD of a pair of general rectangular matrices.
?ggsvp

Computes the preprocessing decomposition for the generalized SVD (deprecated).

Syntax

```c
lapack_int LAPACKE_sggsvp(int matrix_layout, char jobu, char jobv, char jobq,
                         lapack_int m, lapack_int p, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float tola, float tolb, lapack_int* k, lapack_int* l, float* u, lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq);
```

```c
lapack_int LAPACKE_dggsvp(int matrix_layout, char jobu, char jobv, char jobq,
                         lapack_int m, lapack_int p, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double tola, double tolb, lapack_int* k, lapack_int* l, double* u, lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq);
```

```c
lapack_int LAPACKE_cggsvp(int matrix_layout, char jobu, char jobv, char jobq,
                         lapack_int m, lapack_int p, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float tola, float tolb, lapack_int* k, lapack_int* l, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* v, lapack_int ldv, lapack_complex_float* q, lapack_int ldq);
```

```c
lapack_int LAPACKE_zggsvp(int matrix_layout, char jobu, char jobv, char jobq,
                         lapack_int m, lapack_int p, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, double tola, double tolb, lapack_int* k, lapack_int* l, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* v, lapack_int ldv, lapack_complex_double* q, lapack_int ldq);
```

Include Files

- mkl.h

Description

This routine is deprecated; use ggsvp3.

The routine computes orthogonal matrices $U$, $V$ and $Q$ such that

$$
U^H A Q = egin{cases}
  \begin{pmatrix}
    k & 0 & A_{12} & A_{13} \\
    0 & 0 & A_{23}
  \end{pmatrix}, & \text{if } m - k - l > 0 \\
  \begin{pmatrix}
    0 & A_{12} & A_{13} \\
    0 & 0 & A_{23}
  \end{pmatrix}, & \text{if } m - k - l < 0
  \end{cases}
$$

$$
V^H B Q = \begin{pmatrix}
  l & 0 & B_{13} \\
  0 & 0 & 0
  \end{pmatrix}
$$
where the $k$-by-$k$ matrix $A_{12}$ and $l$-by-$l$ matrix $B_{13}$ are nonsingular upper triangular; $A_{23}$ is $l$-by-$l$ upper triangular if $m-k-l \geq 0$, otherwise $A_{23}$ is $(m-k)$-by-$l$ upper trapezoidal. The sum $k+l$ is equal to the effective numerical rank of the $(m+p)$-by-$n$ matrix $(A^H, B^H)^H$.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see subroutine ?tgsja.

**Input Parameters**

*matrix_layout* Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

*jobu* Must be ‘U’ or ‘N’.

  - If *jobu* = ‘U’, orthogonal/unitary matrix $U$ is computed.
  - If *jobu* = ‘N’, $U$ is not computed.

*jobv* Must be ‘V’ or ‘N’.

  - If *jobv* = ‘V’, orthogonal/unitary matrix $V$ is computed.
  - If *jobv* = ‘N’, $V$ is not computed.

*jobq* Must be ‘Q’ or ‘N’.

  - If *jobq* = ‘Q’, orthogonal/unitary matrix $Q$ is computed.
  - If *jobq* = ‘N’, $Q$ is not computed.

*m* The number of rows of the matrix $A$ ($m \geq 0$).

*p* The number of rows of the matrix $B$ ($p \geq 0$).

*n* The number of columns of the matrices $A$ and $B$ ($n \geq 0$).

*a, b* Arrays:

  - $a$ (size at least $\max(1, \text{lda} \times n)$ for column major layout and $\max(1, \text{lda} \times m)$ for row major layout) contains the $m$-by-$n$ matrix $A$.
  - $b$ (size at least $\max(1, \text{ldb} \times n)$ for column major layout and $\max(1, \text{ldb} \times p)$ for row major layout) contains the $p$-by-$n$ matrix $B$.

*lda* The leading dimension of $a$; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

*ldb* The leading dimension of $b$; at least $\max(1, p)$ for column major layout and $\max(1, n)$ for row major layout.

*tola, tolb* $tola$ and $tolb$ are the thresholds to determine the effective numerical rank of matrix $B$ and a subblock of $A$. Generally, they are set to

  - $tola = \max(m, n) \times \|A\| \times \text{MACHEPS}$,
  - $tolb = \max(p, n) \times \|B\| \times \text{MACHEPS}$.

The size of $tola$ and $tolb$ may affect the size of backward errors of the decomposition.

*ldu* The leading dimension of the output array $U$. $ldu \geq \max(1, m)$ if *jobu* = ‘U’; $ldu \geq 1$ otherwise.
The leading dimension of the output array $v$. $ldv \geq \max(1, p)$ if $jobv = 'V' ; ldv \geq 1$ otherwise.

The leading dimension of the output array $q$. $ldq \geq \max(1, n)$ if $jobq = 'Q' ; ldq \geq 1$ otherwise.

Output Parameters

- $a$  
  Overwritten by the triangular (or trapezoidal) matrix described in the Description section.

- $b$  
  Overwritten by the triangular matrix described in the Description section.

- $k$, $l$  
  On exit, $k$ and $l$ specify the dimension of subblocks. The sum $k + l$ is equal to effective numerical rank of $(A^H, B^H)^H$.

- $u$, $v$, $q$  
  Arrays:
  
  - If $jobu = 'U'$, $u$ (size max(1, $ldu$*$m$)) contains the orthogonal/unitary matrix $U$.
  - If $jobu = 'N'$, $u$ is not referenced.
  - If $jobv = 'V'$, $v$ (size max(1, $ldv$*$p$)) contains the orthogonal/unitary matrix $V$.
  - If $jobv = 'N'$, $v$ is not referenced.
  - If $jobq = 'Q'$, $q$ (size max(1, $ldq$*$n$)) contains the orthogonal/unitary matrix $Q$.
  - If $jobq = 'N'$, $q$ is not referenced.

Return Values

This function returns a value $info$.

- If $info=0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.

$\text{ggsvp3}$

Performs preprocessing for a generalized SVD.

Syntax

```c
lapack_int LAPACKE_sggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, float * a, lapack_int lda, float * b,
lapack_int ldb, float tola, float tolb, lapack_int * k, lapack_int * l, float * u,
lapack_int ldu, float * v, lapack_int ldv, float * q, lapack_int ldq);
```

```c
lapack_int LAPACKE_dggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, double * a, lapack_int lda, double * b,
lapack_int ldb, double tola, double tolb, lapack_int * k, lapack_int * l, double * u,
lapack_int ldu, double * v, lapack_int ldv, double * q, lapack_int ldq);
```

```c
lapack_int LAPACKE_cggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, lapack_complex_float * u, lapack_int ldu,
lapack_complex_float * v, lapack_int ldv, lapack_complex_float * q, lapack_int ldq);
```
lapack_int LAPACKE_zggsvp3 (int matrix_layout, char jobu, char jobv, char jobq,  
lapack_int m, lapack_int p, lapack_int n, lapack_complex_double * a, lapack_int lda,  
lapack_complex_double * b, lapack_int ldb, double tola, double tolb, lapack_int * k,  
lapack_int * l, lapack_complex_double * u, lapack_int ldu, lapack_complex_double * v,  
lapack_int ldv, lapack_complex_double * q, lapack_int ldq);

Include Files
• mkl_lapack.h

Include Files
• mkl.h

Description
?ggsvp3 computes orthogonal or unitary matrices \( U, V, \) and \( Q \) such that for real flavors:

For \( \text{m} - \text{k} - \text{l} \geq 0 \):

\[
U^T AQ = \begin{pmatrix}
A_{12} & A_{13} \\
0 & A_{23}
\end{pmatrix}
\]

For \( \text{m} - \text{k} - \text{l} < 0 \):

\[
U^T AQ = \begin{pmatrix}
0 & A_{12} & A_{13} \\
0 & 0 & A_{23}
\end{pmatrix}
\]

\[
V^T BQ = \begin{pmatrix}
0 & 0 & B_{13} \\
0 & 0 & 0
\end{pmatrix}
\]

For complex flavors:

For \( \text{m} - \text{k} - \text{l} \geq 0 \):

\[
U^H AQ = \begin{pmatrix}
A_{12} & A_{13} \\
0 & A_{23}
\end{pmatrix}
\]

For \( \text{m} - \text{k} - \text{l} < 0 \):

\[
U^H AQ = \begin{pmatrix}
0 & A_{12} & A_{13} \\
0 & 0 & A_{23}
\end{pmatrix}
\]

\[
V^H BQ = \begin{pmatrix}
0 & 0 & B_{13} \\
0 & 0 & 0
\end{pmatrix}
\]

where the \( k \)-by-\( k \) matrix \( A_{12} \) and \( l \)-by-\( l \) matrix \( B_{13} \) are nonsingular upper triangular; \( A_{23} \) is \( l \)-by-\( l \) upper triangular if \( \text{m} - \text{k} - \text{l} \geq 0 \), otherwise \( A_{23} \) is \( (m-k) \)-by-\( l \) upper trapezoidal. \( k + l \) is the effective numerical rank of the \( (m+p) \)-by-\( n \) matrix \( (A^T,B^T)^T \) for real flavors or \( (A^H,B^H)^H \) for complex flavors.

This decomposition is the preprocessing step for computing the Generalized Singular Value Decomposition (GSVD), see ?ggsvd3.
Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobu
= 'U': Orthogonal/unitary matrix $U$ is computed;
= 'N': $U$ is not computed.

jobv
= 'V': Orthogonal/unitary matrix $V$ is computed;
= 'N': $V$ is not computed.

jobq
= 'Q': Orthogonal/unitary matrix $Q$ is computed;
= 'N': $Q$ is not computed.

m
The number of rows of the matrix $A$.
$m \geq 0$.

p
The number of rows of the matrix $B$.
$p \geq 0$.

n
The number of columns of the matrices $A$ and $B$.
$n \geq 0$.

a
Array, size ($lda*n$).
On entry, the $m$-by-$n$ matrix $A$.

lda
The leading dimension of the array $a$.
$lda \geq \max(1,m)$.

b
Array, size ($ldb*n$).
On entry, the $p$-by-$n$ matrix $B$.

ldb
The leading dimension of the array $b$.
$ldb \geq \max(1,p)$.

tola, tolb
tola and tolb are the thresholds to determine the effective numerical rank of matrix $B$ and a subblock of $A$. Generally, they are set to

tola = $\max(m,n)\times\text{norm}(a)\times\text{MACHEPS}$,
tolb = $\max(p,n)\times\text{norm}(b)\times\text{MACHEPS}$.

The size of tola and tolb may affect the size of backward errors of the decomposition.

ldu
The leading dimension of the array $u$.
$ldu \geq \max(1,m)$ if $jobu = 'U' $; $ldu \geq 1$ otherwise.

ldv
The leading dimension of the array $v$.
$ldv \geq \max(1,p)$ if $jobv = 'V' $; $ldv \geq 1$ otherwise.

ldq
The leading dimension of the array $q$.
$ldq \geq \max(1,n)$ if $jobq = 'Q' $; $ldq \geq 1$ otherwise.
Output Parameters

**a**
On exit, $a$ contains the triangular (or trapezoidal) matrix described in the Description section.

**b**
On exit, $b$ contains the triangular matrix described in the Description section.

**k, l**
On exit, $k$ and $l$ specify the dimension of the subblocks described in Description section.

$k + l = \text{effective numerical rank of } (A^T,B^T)^T \text{ for real flavors or } (A^H,B^H)^H \text{ for complex flavors.}$

**u**
Array, size ($ldu*m$).
If $jobu = 'U'$, $u$ contains the orthogonal/unitary matrix $U$.
If $jobu = 'N'$, $u$ is not referenced.

**v**
Array, size ($ldv*p$).
If $jobv = 'V'$, $v$ contains the orthogonal/unitary matrix $V$.
If $jobv = 'N'$, $v$ is not referenced.

**q**
Array, size ($ldq*n$).
If $jobq = 'Q'$, $q$ contains the orthogonal/unitary matrix $Q$.
If $jobq = 'N'$, $q$ is not referenced.

Return Values
This function returns a value $info$.

$= 0$: successful exit.
$< 0$: if $info = -i$, the $i$-th argument had an illegal value.

Application Notes

The subroutine uses LAPACK subroutine ?geqp3 for the QR factorization with column pivoting to detect the effective numerical rank of the $A$ matrix. It may be replaced by a better rank determination strategy.

?ggsvp3 replaces the deprecated subroutine ?ggsvp.

?ggsvd3
Computes generalized SVD.

Syntax

```c
lapack_int LAPACKE_sggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
            lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l, float * a,
            lapack_int lda, float * b, lapack_int ldb, float * alpha, float * beta, float * u,
            lapack_int ldu, float * v, lapack_int ldv, float * q, lapack_int ldq, lapack_int * iwork);

lapack_int LAPACKE_dggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
            lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l, double * a,
            lapack_int lda, double * b, lapack_int ldb, double * alpha, double * beta, double * u,
            lapack_int ldu, double * v, lapack_int ldv, double * q, lapack_int ldq, lapack_int * iwork);
```
lapack_int LAPACKE_cggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb,
float * alpha, float * beta, lapack_complex_float * u, lapack_int ldu,
lapack_complex_float * v, lapack_int ldv, lapack_complex_float * q, lapack_int ldq,
lapack_int * iwork);

lapack_int LAPACKE_zggsvd3 (int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int * k, lapack_int * l,
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb,
double * alpha, double * beta, lapack_complex_double * u, lapack_int ldu,
lapack_complex_double * v, lapack_int ldv, lapack_complex_double * q, lapack_int ldq,
lapack_int * iwork);

Include Files
• mkl.h

Description
?ggsvd3 computes the generalized singular value decomposition (GSVD) of an \( m \)-by-\( n \) real or complex matrix \( A \) and \( p \)-by-\( n \) real or complex matrix \( B \):
\[
U^T A Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, \quad V^T B Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix}
\]
for real flavors
or
\[
U^H A Q = D_1 \begin{pmatrix} 0 & R \end{pmatrix}, \quad V^H B Q = D_2 \begin{pmatrix} 0 & R \end{pmatrix}
\]
for complex flavors
where \( U, V \) and \( Q \) are orthogonal/unitary matrices.
Let \( k+l \) be the effective numerical rank of the matrix \((A^T B^T)^T\) for real flavors or the matrix \((A^H B^H)^H\) for complex flavors, then \( R \) is a \((k+l)\)-by-\((k+l)\) nonsingular upper triangular matrix, \( D_1 \) and \( D_2 \) are \( m \)-by-\((k+l)\) and \( p \)-by-\((k+l)\) "diagonal" matrices and of the following structures, respectively:

If \( m-k-l \geq 0 \),
\[
D_1 = \begin{pmatrix} k & l \\
K & L \\
\end{pmatrix}
\begin{pmatrix} I & 0 \\
0 & C \\
\end{pmatrix}
\begin{pmatrix} m-k-l & 0 \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix} k & l \\
K & L \\
\end{pmatrix}
\]
\[
D_2 = \begin{pmatrix} l & 0 \\
p-l & 0 \\
\end{pmatrix}
\begin{pmatrix} 0 & S \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix} 0 & R \\
0 & 0 \\
\end{pmatrix}
\begin{pmatrix} 0 & R \end{pmatrix}
\]
\[
= k \begin{pmatrix} 0 & R_{11} & R_{12} \\
0 & 0 & R_{22} \\
\end{pmatrix}
\]
where
\[
C = \text{diag}(\alpha(k+1), \ldots, \alpha(k+l)),
\]
\[
S = \text{diag}(\beta(k+1), \ldots, \beta(k+l)),
\]
\[
C^2 + S^2 = I.
\]
If \( m-k-l < 0 \),
\[ D_1 = \begin{bmatrix} \frac{k}{m-k} & \frac{k + l - m}{m-k} \\ \frac{1}{0} & C \end{bmatrix} \]

\[ D_2 = \begin{bmatrix} \frac{m - k}{k + l - m} & 0 & \frac{l}{m - k} \\ \frac{0}{0} & S \end{bmatrix} \]

\[
\begin{pmatrix} 0 \\ R \end{pmatrix} = \begin{bmatrix} k & 0 & R_{11} & R_{12} & R_{13} \\ m - k & 0 & R_{22} & R_{23} \\ k + l - m & 0 & 0 & R_{33} \end{bmatrix}
\]

where

\[ C = \text{diag}(\alpha(k + 1), \ldots, \alpha(m)), \]

\[ S = \text{diag}(\beta(k + 1), \ldots, \beta(m)), \]

\[ C^2 + S^2 = I. \]

The routine computes \( C, S, R, \) and optionally the orthogonal/unitary transformation matrices \( U, V \) and \( Q \).

In particular, if \( B \) is an \( n \)-by-\( n \) nonsingular matrix, then the GSVD of \( A \) and \( B \) implicitly gives the SVD of \( A^*\text{inv}(B) \):

\[ A^*\text{inv}(B) = U^*(D_1^*\text{inv}(D_2))*V^T \]

for real flavors

or

\[ A^*\text{inv}(B) = U^*(D_1^*\text{inv}(D_2))*V^H \]

for complex flavors.

If \((A^T,B^T)^T\) for real flavors or \((A^H,B^H)^H\) for complex flavors has orthonormal columns, then the GSVD of \( A \) and \( B \) is also equal to the CS decomposition of \( A \) and \( B \). Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

\[ A^T*AX = \lambda^* B^T*BX \]

for real flavors

or

\[ A^H*AX = \lambda^* B^H*BX \]

for complex flavors.

In some literature, the GSVD of \( A \) and \( B \) is presented in the form

\[ U^*A*X = (0 \ D_1), \ V^*B*X = (0 \ D_2) \]

for real \((A, B)\)

or

\[ U^H*A*X = (0 \ D_1), \ V^H*B*X = (0 \ D_2) \]

for complex \((A, B)\)

where \( U \) and \( V \) are orthogonal and \( X \) is nonsingular, \( D_1 \) and \( D_2 \) are "diagonal". The former GSVD form can be converted to the latter form by taking the nonsingular matrix \( X \) as

\[ X = Q* \begin{bmatrix} I & 0 \\ 0 & \text{inv}(R) \end{bmatrix} \]

**Input Parameters**

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobu**:
  - = ‘U’: Orthogonal/unitary matrix \( U \) is computed;
  - = ‘N’: \( U \) is not computed.
jobv
= 'V': Orthogonal/unitary matrix V is computed;
= 'N': V is not computed.

jobq
= 'Q': Orthogonal/unitary matrix Q is computed;
= 'N': Q is not computed.

m
The number of rows of the matrix A.
\( m \geq 0 \).

n
The number of columns of the matrices A and B.
\( n \geq 0 \).

p
The number of rows of the matrix B.
\( p \geq 0 \).

a
Array, size \((lda*n)\).
On entry, the \( m \)-by-\( n \) matrix A.

lda
The leading dimension of the array a.
\( lda \geq \max(1,m) \).

b
Array, size \((ldb*n)\).
On entry, the \( p \)-by-\( n \) matrix B.

ldb
The leading dimension of the array b.
\( ldb \geq \max(1,p) \).

ldu
The leading dimension of the array u.
\( ldu \geq \max(1,m) \) if jobu = 'U'; \( ldu \geq 1 \) otherwise.

ldv
The leading dimension of the array v.
\( ldv \geq \max(1,p) \) if jobv = 'V'; \( ldv \geq 1 \) otherwise.

ldq
The leading dimension of the array q.
\( ldq \geq \max(1,n) \) if jobq = 'Q'; \( ldq \geq 1 \) otherwise.

iwork
Array, size \((n)\).

Output Parameters

k, l
On exit, \( k \) and \( l \) specify the dimension of the subblocks described in the Description section.
\[ k + l \text{ is effective numerical rank of } (A^T,B^T)^T \text{ for real flavors or } (A^H,B^H)^H \text{ for complex flavors}. \]

a
On exit, \( a \) contains the triangular matrix \( R \), or part of \( R \).
If \( m-k-l \geq 0 \), \( R \) is stored in the elements of array \( a \) corresponding to \( A_1: \\
\phantom{=} k + l,n - k - l + 1:n. \]
If \( m - k - l < 0 \), \[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} \\
0 & R_{22} & R_{23}
\end{bmatrix}
\] is stored in the elements of array \( a \) corresponding to \( A(1:m, n - k - l + 1:n) \), and \( R_{33} \) is stored in the elements of array \( a \) corresponding to \( A(m - k + 1:1, n + m - k - l + 1:n) \) on exit.

On exit, \( b \) contains part of the triangular matrix \( R \) if \( m - k - l < 0 \). See Description for details.

- **alpha**: Array, size \((n)\)
  
  On exit, \( \alpha \) and \( \beta \) contain the generalized singular value pairs of \( a \) and \( b \):
  
  \[
  \alpha[0: k - 1] = 1, \\
  \beta[0: k - 1] = 0,
  \]
  
  and if \( m - k - l \geq 0 \),
  
  \[
  \alpha[k:k + l - 1] = C, \\
  \beta[k:k + l - 1] = S,
  \]
  
  or if \( m - k - l < 0 \),
  
  \[
  \alpha[k:m - 1] = C, \alpha[m: k + l - 1] = 0 \\
  \beta[k: m - 1] = S, \beta[m: k + l - 1] = 1
  \]
  
  and
  
  \[
  \alpha[k + l: n - 1] = 0 \\
  \beta[k + l : n - 1] = 0
  \]

- **u**: Array, size \((ldu*m)\).
  
  If \( jobu = 'U' \), \( u \) contains the \( m \)-by-\( m \) orthogonal/unitary matrix \( U \).
  If \( jobu = 'N' \), \( u \) is not referenced.

- **v**: Array, size \((ldv*p)\).
  
  If \( jobv = 'V' \), \( v \) contains the \( p \)-by-\( p \) orthogonal/unitary matrix \( V \).
  If \( jobv = 'N' \), \( v \) is not referenced.

- **q**: Array, size \((ldq*n)\).
  
  If \( jobq = 'Q' \), \( q \) contains the \( n \)-by-\( n \) orthogonal/unitary matrix \( Q \).
  If \( jobq = 'N' \), \( q \) is not referenced.

- **iwork**: On exit, \( iwork \) stores the sorting information. More precisely, the following loop uses \( iwork \) to sort \( \alpha \):
  
  ```
  for (i = k; i<min(m,k+l); i++) {
      swap (alpha[i], alpha[iwork[i] - 1]);
  }
  ```

  such that \( \alpha[0] \geq \alpha[1] \geq ... \geq \alpha[n-1] \).
Return Values

This function returns a value info.

= 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.
> 0: if info = 1, the Jacobi-type procedure failed to converge.

For further details, see subroutine ?tgsja.

Application Notes

?ggsvd3 replaces the deprecated subroutine ?gsvd.

?tgsja

Computes the generalized SVD of two upper triangular or trapezoidal matrices.

Syntax

lapack_int LAPACKE_stgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int l, float* a,
lapack_int lda, float* b, lapack_int ldb, float tola, float tolb, float* alpha, float* beta,
float* u, lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq,
lapack_int* ncycle);

lapack_int LAPACKE_dtgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int l, double* a,
lapack_int lda, double* b, lapack_int ldb, double tola, double tolb, double* alpha,
double* beta, double* u, lapack_int ldu, double* v, lapack_int ldv, double* q,
lapack_int ldq, lapack_int* ncycle);

lapack_int LAPACKE_ctgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int l,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
float tola, float tolb, float* alpha, float* beta, lapack_complex_float* u,
lapack_complex_float* v, lapack_int ldu, lapack_complex_float* q, lapack_int ldq,
lapack_int* ncycle);

lapack_int LAPACKE_ztgsja( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int p, lapack_int n, lapack_int l,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
double tola, double tolb, double* alpha, double* beta, lapack_complex_double* u,
lapack_complex_double* v, lapack_int ldu, lapack_complex_double* q,
lapack_int ldq, lapack_int* ncycle);

Include Files

• mkl.h

Description

The routine computes the generalized singular value decomposition (GSVD) of two real/complex upper triangular (or trapezoidal) matrices A and B. On entry, it is assumed that matrices A and B have the following forms, which may be obtained by the preprocessing subroutine ggsvp from a general m-by-n matrix A and p-by-n matrix B:
where the \( k \)-by-\( k \) matrix \( A_{12} \) and \( l \)-by-\( l \) matrix \( B_{13} \) are nonsingular upper triangular; \( A_{23} \) is \( l \)-by-\( l \) upper triangular if \( m-k-l \geq 0 \), otherwise \( A_{23} \) is \((m-k)\)-by-\( l \) upper trapezoidal.

On exit,
\[
U^*A^*Q = D_1^*(0 R), \quad V^*B^*Q = D_2^*(0 R),
\]
where \( U, V \) and \( Q \) are orthogonal/unitary matrices, \( R \) is a nonsingular upper triangular matrix, and \( D_1 \) and \( D_2 \) are "diagonal" matrices, which are of the following structures:

If \( m-k-l \geq 0 \),
\[
D_1 = \begin{pmatrix} k & l \\ \end{pmatrix}, \quad D_2 = \begin{pmatrix} 0 & S \\ \end{pmatrix}
\]

where
\[
C = \text{diag}(\alpha[k], \ldots, \alpha[k+l-1])
\]
\[
S = \text{diag}(\beta[k], \ldots, \beta[k+l-1])
\]
\[ C^2 + S^2 = I \]

\( R \) is stored in \( a(1:k+l, n-k-l+1:n) \) on exit.

If \( m-k-l < 0 \),

\[
\begin{bmatrix}
  k & m-k & k+1-m \\
  m-k & C & 0 \\
\end{bmatrix}
\]

\[
D_2 = \begin{bmatrix}
  m-k & S & 0 \\
  k+l-m & 0 & 0 \\
  p-l & 0 & 0 \\
\end{bmatrix}
\]

\[
(0 \ P) = \begin{bmatrix}
  k & 0 & R_{11} & R_{12} & R_{13} \\
  m-k & 0 & R_{22} & R_{23} \\
  k+l-m & 0 & 0 & 0 & R_{33} \\
\end{bmatrix}
\]

where

\[
C = \text{diag}(\alpha[k], \ldots, \alpha[m-1]),
\]

\[
S = \text{diag}(\beta[k], \ldots, \beta[m-1]),
\]

\[ C^2 + S^2 = I \]

\[
\begin{pmatrix}
  R_{11} & R_{12} & R_{13} \\
  0 & R_{22} & R_{23} \\
\end{pmatrix}
\]

On exit, \( (0 \ P) \) is stored in \( a(1:m, n-k-l+1:n) \) and \( R_{33} \) is stored in \( b(m-k+1:l, n+m-k-l+1:n) \).

The computation of the orthogonal/unitary transformation matrices \( U, V \) or \( Q \) is optional. These matrices may either be formed explicitly, or they may be postmultiplied into input matrices \( U_1, V_1, \) or \( Q_1 \).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobu**
  Must be 'U', 'I', or 'N'.
  - If \( \text{jobu} = 'U' \), \( u \) must contain an orthogonal/unitary matrix \( U_1 \) on entry.
  - If \( \text{jobu} = 'I' \), \( u \) is initialized to the unit matrix.
  - If \( \text{jobu} = 'N' \), \( u \) is not computed.

- **jobv**
  Must be 'V', 'I', or 'N'.
  - If \( \text{jobv} = 'V' \), \( v \) must contain an orthogonal/unitary matrix \( V_1 \) on entry.
  - If \( \text{jobv} = 'I' \), \( v \) is initialized to the unit matrix.
If \( \textit{jobv} = 'N' \), \( v \) is not computed.

\textbf{jobq}

Must be 'Q', 'I', or 'N'.

If \( \textit{jobq} = 'Q' \), \( q \) must contain an orthogonal/unitary matrix \( Q_1 \) on entry.

If \( \textit{jobq} = 'I' \), \( q \) is initialized to the unit matrix.

If \( \textit{jobq} = 'N' \), \( q \) is not computed.

\textbf{m}

The number of rows of the matrix \( A \) (\( m \geq 0 \)).

\textbf{p}

The number of rows of the matrix \( B \) (\( p \geq 0 \)).

\textbf{n}

The number of columns of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

\textbf{k, l}

Specify the subblocks in the input matrices \( A \) and \( B \), whose GSVD is computed.

\textbf{a, b, u, v, q}

Arrays:

\( a \) (size at least \( \max(1, \text{lda} \times n) \) for column major layout and \( \max(1, \text{lda} \times m) \) for row major layout) contains the \( m \)-by-\( n \) matrix \( A \).

\( b \) (size at least \( \max(1, \text{ldb} \times n) \) for column major layout and \( \max(1, \text{ldb} \times p) \) for row major layout) contains the \( p \)-by-\( n \) matrix \( B \).

If \( \textit{jobu} = 'U' \), \( u \) (size \( \max(1, \text{ldu} \times m) \)) must contain a matrix \( U_1 \) (usually the orthogonal/unitary matrix returned by \(?\text{ggsvp}\)).

If \( \textit{jobv} = 'V' \), \( v \) (size \( \max(1, \text{ldv} \times p) \)) must contain a matrix \( V_1 \) (usually the orthogonal/unitary matrix returned by \(?\text{ggsvp}\)).

If \( \textit{jobq} = 'Q' \), \( q \) (size \( \max(1, \text{ldq} \times n) \)) must contain a matrix \( Q_1 \) (usually the orthogonal/unitary matrix returned by \(?\text{ggsvp}\)).

\textbf{lda}

The leading dimension of \( a \); at least \( \max(1, \text{i} \times m) \) for column major layout and \( \max(1, \text{i} \times n) \) for row major layout.

\textbf{ldb}

The leading dimension of \( b \); at least \( \max(1, \text{i} \times p) \) for column major layout and \( \max(1, \text{i} \times n) \) for row major layout.

\textbf{ldu}

The leading dimension of the array \( u \).

\( \text{ldu} \geq \max(1, \text{i}) \) if \( \textit{jobu} = 'U'; \text{ldu} \geq 1 \) otherwise.

\textbf{ldv}

The leading dimension of the array \( v \).

\( \text{ldv} \geq \max(1, \text{i}) \) if \( \textit{jobv} = 'V'; \text{ldv} \geq 1 \) otherwise.

\textbf{ldq}

The leading dimension of the array \( q \).

\( \text{ldq} \geq \max(1, \text{i}) \) if \( \textit{jobq} = 'Q'; \text{ldq} \geq 1 \) otherwise.

\textbf{tola, tolb}

tola and tolb are the convergence criteria for the Jacobi-Kogbetliantz iteration procedure. Generally, they are the same as used in \(?\text{ggsvp}\):

\( \text{tola} = \max(\text{m}, \text{n}) \times |A| \times \text{MACHEPS}, \)

\( \text{tolb} = \max(\text{p}, \text{n}) \times |B| \times \text{MACHEPS}. \)
Output Parameters

\( a \)

On exit, \( a(n-k+1:n, 1:\min(k+l, m)) \) contains the triangular matrix \( R \) or part of \( R \).

\( b \)

On exit, if necessary, \( b(m-k+1: l, n+m-k-l+1: n) \) contains a part of \( R \).

\( \alpha, \beta \)

Arrays, size at least \( \max(1,n) \). Contain the generalized singular value pairs of \( A \) and \( B \):

\[
\alpha(1:k) = 1, \\
\beta(1:k) = 0, \\
\text{and if } m-k-l \geq 0, \\
\alpha(k+1:k+l) = \text{diag}(C), \\
\beta(k+1:k+l) = \text{diag}(S), \\
or if m-k-l < 0, \\
\alpha(k+1:m) = \text{diag}(C), \alpha(m+1:k+l) = 0 \\
\beta(k+1:m) = \text{diag}(S), \\
\beta(m+1:k+l) = 1.
\]

Furthermore, if \( k+l < n \),

\[
\alpha(k+l+1:n) = 0 \text{ and} \\
\beta(k+l+1:n) = 0.
\]

\( u \)

If \( \text{jobu} = 'I' \), \( u \) contains the orthogonal/unitary matrix \( U \).

If \( \text{jobu} = 'U' \), \( u \) contains the product \( U_1 * U \).

If \( \text{jobu} = 'N' \), \( u \) is not referenced.

\( v \)

If \( \text{jobv} = 'I' \), \( v \) contains the orthogonal/unitary matrix \( U \).

If \( \text{jobv} = 'V' \), \( v \) contains the product \( V_1 * V \).

If \( \text{jobv} = 'N' \), \( v \) is not referenced.

\( q \)

If \( \text{jobq} = 'I' \), \( q \) contains the orthogonal/unitary matrix \( U \).

If \( \text{jobq} = 'Q' \), \( q \) contains the product \( Q_1 * Q \).

If \( \text{jobq} = 'N' \), \( q \) is not referenced.

\( ncycle \)

The number of cycles required for convergence.

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = 1 \), the procedure does not converge after \( \text{MAXIT} \) cycles.
Cosine-Sine Decomposition: LAPACK Computational Routines

This section describes LAPACK computational routines for computing the *cosine-sine decomposition* (CS decomposition) of a partitioned unitary/orthogonal matrix. The algorithm computes a complete 2-by-2 CS decomposition, which requires simultaneous diagonalization of all the four blocks of a unitary/orthogonal matrix partitioned into a 2-by-2 block structure.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Computational Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real matrices</th>
<th>Complex matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form</td>
<td>bbcsd/bbcsd</td>
<td>bbcsd/bbcsd</td>
</tr>
<tr>
<td>Simultaneously bidiagonalize the blocks of a partitioned orthogonal matrix</td>
<td>orbdb unbdb</td>
<td>orbdb unbdb</td>
</tr>
<tr>
<td>Simultaneously bidiagonalize the blocks of a partitioned unitary matrix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

CS Driver Routine

?bbcsd

*Computes the CS decomposition of an orthogonal/unitary matrix in bidiagonal-block form.*

**Syntax**

```c
lapack_int LAPACKE_sbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float* phi, float* u1, lapack_int ldu1, float* u2, lapack_int ldu2, float* v1t, lapack_int ldv1t, float* v2t, lapack_int ldv2t, float* b11d, float* b11e, float* b12d, float* b12e, float* b21d, float* b21e, float* b22d, float* b22e );

lapack_int LAPACKE_dbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double* phi, double* u1, lapack_int ldu1, double* u2, lapack_int ldu2, double* v1t, lapack_int ldv1t, double* v2t, lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double* b21d, double* b21e, double* b22d, double* b22e );

lapack_int LAPACKE_cbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, float* theta, float* phi, lapack_complex_float* u1, lapack_int ldu1, lapack_complex_float* u2, lapack_int ldu2, lapack_complex_float* v1t, lapack_int ldv1t, lapack_complex_float* v2t, lapack_int ldv2t, float* b11d, float* b11e, float* b12d, float* b12e, float* b21d, float* b21e, float* b22d, float* b22e );

lapack_int LAPACKE_zbbcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, lapack_int m, lapack_int p, lapack_int q, double* theta, double* phi, lapack_complex_double* u1, lapack_int ldu1, lapack_complex_double* u2, lapack_int ldu2, lapack_complex_double* v1t, lapack_int ldv1t, lapack_complex_double* v2t, lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double* b21d, double* b21e, double* b22d, double* b22e );
```
ldu2, lapack_complex_double* v1t, lapack_int ldv1t, lapack_complex_double* v2t, lapack_int ldv2t, double* b11d, double* b11e, double* b12d, double* b12e, double* b21d, double* b21e, double* b22d, double* b22e );

Include Files
• mkl.h

Description
mkl_lapack.fi The routine bbbcsd computes the CS decomposition of an orthogonal or unitary matrix in bidiagonal-block form:

\[
X = \begin{pmatrix}
    b_{11} & b_{12} & 0 & 0 \\
    0 & 0 & -I & 0 \\
    \xi_21 & \xi_22 & 0 & 0 \\
    0 & 0 & 0 & I \\
\end{pmatrix} = \begin{pmatrix}
    u_1 & | & u_2 \\
    | & | & | \\
C & | & -S & 0 & 0 \\
0 & 0 & -I & 0 \\
S & | & C & 0 & 0 \\
0 & 0 & 0 & I \\
\end{pmatrix} \begin{pmatrix}
    v_1 \\
v_2 \\
\end{pmatrix}^T
\]

or

\[
X = \begin{pmatrix}
    b_{11} & b_{12} & 0 & 0 \\
0 & 0 & -I & 0 \\
\xi_21 & \xi_22 & 0 & 0 \\
0 & 0 & 0 & I \\
\end{pmatrix} = \begin{pmatrix}
    u_1 & | & u_2 \\
    | & | & | \\
C & | & -S & 0 & 0 \\
0 & 0 & -I & 0 \\
S & | & C & 0 & 0 \\
0 & 0 & 0 & I \\
\end{pmatrix} \begin{pmatrix}
    v_1 \\
v_2 \\
\end{pmatrix}^H
\]

respectively.

\( x \) is \( m \)-by-\( m \) with the top-left block \( p \)-by-\( q \). Note that \( q \) must not be larger than \( p, m-p, \) or \( m-q \). If \( q \) is not the smallest index, \( x \) must be transposed and/or permuted in constant time using the \( \text{trans} \) option. See ?orcsd/?uncsd for details.

The bidiagonal matrices \( b_{11}, b_{12}, b_{21}, \) and \( b_{22} \) are represented implicitly by angles \( \text{theta}(1:q) \) and \( \text{phi}(1:q-1) \).

The orthogonal/unitary matrices \( u_1, u_2, v_{1t}, \) and \( v_{2t} \) are input/output. The input matrices are pre- or post-multiplied by the appropriate singular vector matrices.

Input Parameters

\text{matrix\_layout} \hspace{2cm} \text{Specifies whether matrix storage layout is row major (LAPACK\_ROW\_MAJOR) or column major (LAPACK\_COL\_MAJOR).}

\text{jobu1} \hspace{2cm} \text{If equals Y, then } u_1 \text{ is updated. Otherwise, } u_1 \text{ is not updated.}

\text{jobu2} \hspace{2cm} \text{If equals Y, then } u_2 \text{ is updated. Otherwise, } u_2 \text{ is not updated.}

\text{jobv1t} \hspace{2cm} \text{If equals Y, then } v_{1t} \text{ is updated. Otherwise, } v_{1t} \text{ is not updated.}

\text{jobv2t} \hspace{2cm} \text{If equals Y, then } v_{2t} \text{ is updated. Otherwise, } v_{2t} \text{ is not updated.}

\text{trans} \hspace{2cm} \text{If } \text{trans} = 'T', \text{ then } x, u_1, u_2, v_{1t}, v_{2t} \text{ are stored in row-major order. Otherwise, } x, u_1, u_2, v_{1t}, v_{2t} \text{ are stored in column-major order.}

\text{m} \hspace{2cm} \text{The number of rows and columns of the orthogonal/unitary matrix } X \text{ in bidiagonal-block form.}
p

The number of rows in the top-left block of \( x \). \( 0 \leq p \leq m \).

q

The number of columns in the top-left block of \( x \). \( 0 \leq q \leq \min(p, m-p, m-q) \).

theta

Array, size \( q \).

On entry, the angles \( \theta[0], ..., \theta[q-1] \) that, along with \( \phi[0], ..., \phi[q-2] \), define the matrix in bidiagonal-block form as returned by ordbdb/unbdb.

phi

Array, size \( q-1 \).

The angles \( \phi[0], ..., \phi[q-2] \) that, along with \( \theta[0], ..., \theta[q-1] \), define the matrix in bidiagonal-block form as returned by ordbdb/unbdb.

u1

Array, size at least \( \max(1, ldu1 \times p) \).

On entry, a \( p \times p \) matrix.

ldu1

The leading dimension of the array \( u_1 \), \( ldu1 \leq \max(1, p) \).

u2

Array, size \( \max(1, ldu2 \times (m-p)) \).

On entry, an \( (m-p) \times (m-p) \) matrix.

ldu2

The leading dimension of the array \( u_2 \), \( ldu2 \leq \max(1, m-p) \).

v1t

Array, size \( \max(1, ldv1t \times q) \).

On entry, a \( q \times q \) matrix.

ldv1t

The leading dimension of the array \( v_{1t} \), \( ldv1t \leq \max(1, q) \).

v2t

Array, size.

On entry, an \( (m-q) \times (m-q) \) matrix.

ldv2t

The leading dimension of the array \( v_{2t} \), \( ldv2t \leq \max(1, m-q) \).

Output Parameters

theta

On exit, the angles whose cosines and sines define the diagonal blocks in the CS decomposition.

u1

On exit, \( u_1 \) is postmultiplied by the left singular vector matrix common to \([ \begin{bmatrix} b_{11} & 0 \end{bmatrix} \]) and \([ \begin{bmatrix} b_{12} & 0 & 0 \end{bmatrix} ; 0 & -I & 0 \] \).

u2

On exit, \( u_2 \) is postmultiplied by the left singular vector matrix common to \([ \begin{bmatrix} b_{21} & 0 \end{bmatrix} \]) and \([ \begin{bmatrix} b_{22} & 0 & 0 \end{bmatrix} ; 0 & 0 & I \] \).

v1t

Array, size \( q \).

On exit, \( v_{1t} \) is premultiplied by the transpose of the right singular vector matrix common to \([ \begin{bmatrix} 0 \end{bmatrix} ] \) and \([ \begin{bmatrix} b_{11} \end{bmatrix} ; 0 \] \).

v2t

On exit, \( v_{2t} \) is premultiplied by the transpose of the right singular vector matrix common to \([ \begin{bmatrix} b_{12} & 0 & 0 \end{bmatrix} ; 0 & -I & 0 \] \) and \([ \begin{bmatrix} b_{22} & 0 & 0 \end{bmatrix} ; 0 & 0 & I \] \).

b11d

Array, size \( q \).
When \( \text{bbcsd} \) converges, \( b11d \) contains the cosines of \( \theta[0], \ldots, \theta[q-1] \). If \( \text{bbcsd} \) fails to converge, \( b11d \) contains the diagonal of the partially reduced top left block.

\( b11e \)
Array, size \( q-1 \).
When \( \text{bbcsd} \) converges, \( b11e \) contains zeros. If \( \text{bbcsd} \) fails to converge, \( b11e \) contains the superdiagonal of the partially reduced top left block.

\( b12d \)
Array, size \( q \).
When \( \text{bbcsd} \) converges, \( b12d \) contains the negative sines of \( \theta[0], \ldots, \theta[q-1] \). If \( \text{bbcsd} \) fails to converge, \( b12d \) contains the diagonal of the partially reduced top right block.

\( b12e \)
Array, size \( q-1 \).
When \( \text{bbcsd} \) converges, \( b12e \) contains zeros. If \( \text{bbcsd} \) fails to converge, \( b12e \) contains the superdiagonal of the partially reduced top right block.

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info}=0 \), the execution is successful.
- If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.
- If \( \text{info} > 0 \) and if \( \text{bbcsd} \) did not converge, \( \text{info} \) specifies the number of nonzero entries in \( \phi \), and \( b11d, b11e, \) etc. contain the partially reduced matrix.

**See Also**

?orcsl/?uncslxerbla

?orbdb/?unbdb

*Simultaneously bidiagonalizes the blocks of a partitioned orthogonal/unitary matrix.*

**Syntax**

```c
lapack_int LAPACKE_sorbdb( int matrix_layout, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, float* x11, lapack_int ldx11, float* x12, lapack_int ldx12, float* x21, lapack_int ldx21, float* x22, lapack_int ldx22, float* theta, float* phi, float* taup1, float* taup2, float* tauql, float* tauqr );

lapack_int LAPACKE_dorbdb( int matrix_layout, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, double* x11, lapack_int ldx11, double* x12, lapack_int ldx12, double* x21, lapack_int ldx21, double* x22, lapack_int ldx22, double* theta, double* phi, double* taup1, double* taup2, double* tauql, double* tauqr );

lapack_int LAPACKE_cunbdb( int matrix_layout, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, lapack_complex_float* x11, lapack_int ldx11, lapack_complex_float* x12, lapack_int ldx12, lapack_complex_float* x21, lapack_int ldx21, lapack_complex_float* x22, lapack_int ldx22, float* theta, float* phi, lapack_complex_float* taup1, lapack_complex_float* taup2, lapack_complex_float* tauql, lapack_complex_float* tauqr );
```
lapack_int LAPACKE_zunbdb( int matrix_layout, char trans, char signs, lapack_int m,
lapack_int p, lapack_int q, lapack_complex_double* x11, lapack_int ldx11,
lapack_complex_double* x12, lapack_int ldx12, lapack_complex_double* x21, lapack_int
ldx21, lapack_complex_double* x22, lapack_int ldx22, double* theta, double* phi,
lapack_complex_double* taup1, lapack_complex_double* taup2, lapack_complex_double*
tauq1, lapack_complex_double* tauq2 );

Include Files
• mkl.h

Description
The routines ?orbdb/?unbdb simultaneously bidiagonalizes the blocks of an m-by-m partitioned orthogonal
matrix $X$:

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | \\ | & p_2 \end{pmatrix} \begin{pmatrix} b_{11} | b_{12} & 0 & 0 \\ 0 & 0 & -I & 0 \\ b_{21} | b_{22} & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | \\ | & q_2 \end{pmatrix}^T$$

or unitary matrix:

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \begin{pmatrix} p_1 & | \\ | & p_2 \end{pmatrix} \begin{pmatrix} b_{11} | b_{12} & 0 & 0 \\ 0 & 0 & -I & 0 \\ b_{21} | b_{22} & 0 & 0 \\ 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} q_1 & | \\ | & q_2 \end{pmatrix}^H$$

$x_{11}$ is p-by-q. $q$ must not be larger than $p$, $m-p$, or $m-q$. Otherwise, $X$ must be transposed and/or permuted
in constant time using the trans and signs options.

The orthogonal/unitary matrices $p_1, p_2, q_1$, and $q_2$ are p-by-p, (m-p)-by-(m-p), q-by-q, (m-q)-by-(m-q),
respectively. They are represented implicitly by Housholder vectors.

The bidiagonal matrices $b_{11}, b_{12}, b_{21}$, and $b_{22}$ are q-by-q bidiagonal matrices represented implicitly by angles
$\text{theta}[0], \ldots, \text{theta}[\text{q}-1]$ and $\text{phi}[0], \ldots, \text{phi}[\text{q}-2]$. $b_{11}$ and $b_{12}$ are upper bidiagonal, while $b_{21}$ and
$b_{22}$ are lower bidiagonal. Every entry in each bidiagonal band is a product of a sine or cosine of $\text{theta}$ with a
sine or cosine of $\phi$. See [Sutton09] for details.

$p_1, p_2, q_1$, and $q_2$ are represented as products of elementary reflectors.

Input Parameters

`matrix_layout` 
Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

`trans` 
= 'T': $X, u_1, u_2, v_1^t, v_2^t$ are stored in row-major order.
otherwise $X, u_1, u_2, v_1^t, v_2^t$ are stored in column-major order.

`signs` 
= 'O': The lower-left block is made nonpositive (the "other" convention).
otherwise The upper-right block is made nonpositive (the "default" convention).
m

The number of rows and columns of the matrix X.

p

The number of rows in x_{11} and x_{12}. \( 0 \leq p \leq m \).

q

The number of columns in x_{11} and x_{21}. \( 0 \leq q \leq \min(p, m-p, m-q) \).

x_{11}

Array, size (size \( \max(1, ldx_{11} \times q) \) for column major layout and \( \max(1, ldx_{11} \times p) \) for row major layout).

On entry, the top-left block of the orthogonal/unitary matrix to be reduced.

ldx_{11}

The leading dimension of the array X_{11}. If \( \text{trans} = 'T' \), \( ldx_{11} \geq p \) for column major layout and \( ldx_{11} \geq q \) for row major layout. Otherwise, \( ldx_{11} \geq q \).

x_{12}

Array, size (size \( \max(1, ldx_{12} \times (m-q)) \) for column major layout and \( \max(1, ldx_{12} \times p) \) for row major layout).

On entry, the top-right block of the orthogonal/unitary matrix to be reduced.

ldx_{12}

The leading dimension of the array X_{12}. If \( \text{trans} = 'N' \), \( ldx_{12} \geq p \) for column major layout and \( ldx_{12} \geq m-q \) for row major layout. Otherwise, \( ldx_{12} \geq m-q \).

x_{21}

Array, size (size \( \max(1, ldx_{21} \times q) \) for column major layout and \( \max(1, ldx_{21} \times (m-p)) \) for row major layout).

On entry, the bottom-left block of the orthogonal/unitary matrix to be reduced.

ldx_{21}

The leading dimension of the array X_{21}. If \( \text{trans} = 'N' \), \( ldx_{21} \geq m-p \) for column major layout and \( ldx_{21} \geq q \) for row major layout. Otherwise, \( ldx_{21} \geq q \).

x_{22}

Array, size (size \( \max(1, ldx_{22} \times (m-q)) \) for column major layout and \( \max(1, ldx_{22} \times (m-p)) \) for row major layout).

On entry, the bottom-right block of the orthogonal/unitary matrix to be reduced.

ldx_{22}

The leading dimension of the array X_{22}. If \( \text{trans} = 'N' \), \( ldx_{22} \geq m-p \) for column major layout and \( ldx_{22} \geq m-q \) for row major layout. Otherwise, \( ldx_{22} \geq m-q \).

**Output Parameters**

x_{11}

On exit, the form depends on \( \text{trans} \): 

If \( \text{trans}= 'N' \), 
the columns of the lower triangle of x_{11} specify reflectors for p_{1}, the rows of the upper triangle of x_{11}(1:q - 1, q:q - 1) specify reflectors for q_{1}

otherwise

\( \text{trans}= 'T' \),
the rows of the upper triangle of x_{11} specify reflectors for p_{1}, the columns of the lower triangle of x_{11}(1:q - 1, q:q - 1) specify reflectors for q_{1}

x_{12}

On exit, the form depends on \( \text{trans} \):
If \( \text{trans}='N' \), the columns of the upper triangle of \( x_{12} \) specify the first \( p \) reflectors for \( q_2 \)

otherwise \( \text{trans}='T' \), the columns of the lower triangle of \( x_{12} \) specify the first \( p \) reflectors for \( q_2 \)

\( x_{21} \)

On exit, the form depends on \( \text{trans} \):

If \( \text{trans}='N' \), the columns of the lower triangle of \( x_{21} \) specify the reflectors for \( p_2 \)

otherwise \( \text{trans}='T' \), the columns of the upper triangle of \( x_{21} \) specify the reflectors for \( p_2 \)

\( x_{22} \)

On exit, the form depends on \( \text{trans} \):

If \( \text{trans}='N' \), the rows of the upper triangle of \( x_{22}(q+1:m-p,p+1:m-q) \) specify the last \( m-p-q \) reflectors for \( q_2 \)

otherwise \( \text{trans}='T' \), the columns of the lower triangle of \( x_{22}(p+1:m-q,q+1:m-p) \) specify the last \( m-p-q \) reflectors for \( p_2 \)

\( \theta \)

Array, size \( q \). The entries of bidiagonal blocks \( b_{11}, b_{12}, b_{21}, \) and \( b_{22} \) can be computed from the angles \( \theta \) and \( \phi \). See the Description section for details.

\( \phi \)

Array, size \( q-1 \). The entries of bidiagonal blocks \( b_{11}, b_{12}, b_{21}, \) and \( b_{22} \) can be computed from the angles \( \theta \) and \( \phi \). See the Description section for details.

\( \text{taup}_1 \)

Array, size \( p \).

Scalar factors of the elementary reflectors that define \( p_1 \).

\( \text{taup}_2 \)

Array, size \( m-p \).

Scalar factors of the elementary reflectors that define \( p_2 \).

\( \text{tauq}_1 \)

Array, size \( q \).

Scalar factors of the elementary reflectors that define \( q_1 \).

\( \text{tauq}_2 \)

Array, size \( m-q \).

Scalar factors of the elementary reflectors that define \( q_2 \).

Return Values

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

See Also

?orcsd/?uncsd
?orgqr
?ungqr
?orglq
?unglq
xerbla
LAPACK Least Squares and Eigenvalue Problem Driver Routines

Each of the LAPACK driver routines solves a complete problem. To arrive at the solution, driver routines typically call a sequence of appropriate computational routines.

Driver routines are described in the following sections:
Linear Least Squares (LLS) Problems
Generalized LLS Problems
Symmetric Eigenproblems
Nonsymmetric Eigenproblems
Singular Value Decomposition
Cosine-Sine Decomposition
Generalized Symmetric Definite Eigenproblems
Generalized Nonsymmetric Eigenproblems

Linear Least Squares (LLS) Problems: LAPACK Driver Routines

This section describes LAPACK driver routines used for solving linear least squares problems. Table "Driver Routines for Solving LLS Problems" lists all such routines.

Driver Routines for Solving LLS Problems

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gels</td>
<td>Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.</td>
</tr>
<tr>
<td>gelsy</td>
<td>Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of A.</td>
</tr>
<tr>
<td>gelss</td>
<td>Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A.</td>
</tr>
<tr>
<td>gelsd</td>
<td>Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of A and a divide and conquer method.</td>
</tr>
</tbody>
</table>

?gels

Uses QR or LQ factorization to solve a overdetermined or underdetermined linear system with full rank matrix.

Syntax

```c
lapack_int LAPACKE_sgels (int matrix_layout, char trans, lapack_int m, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* b, lapack_int ldb);

lapack_int LAPACKE_dgels (int matrix_layout, char trans, lapack_int m, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb);

lapack_int LAPACKE_cgels (int matrix_layout, char trans, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb);

lapack_int LAPACKE_zgels (int matrix_layout, char trans, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb);
```
Include Files

- mkl.h

Description

The routine solves overdetermined or underdetermined real/complex linear systems involving an \( m\)-by-\( n\) matrix \( A \), or its transpose/conjugate-transpose, using a QR or LQ factorization of \( A \). It is assumed that \( A \) has full rank.

The following options are provided:

1. If \( \text{trans} = 'N' \) and \( m \geq n \): find the least squares solution of an overdetermined system, that is, solve the least squares problem
   \[
   \minimize \|b - A^*x\|_2
   \]
2. If \( \text{trans} = 'N' \) and \( m < n \): find the minimum norm solution of an underdetermined system \( A^*X = B \).
3. If \( \text{trans} = 'T' \) or \( 'C' \) and \( m \geq n \): find the minimum norm solution of an undetermined system \( A^H*X = B \).
4. If \( \text{trans} = 'T' \) or \( 'C' \) and \( m < n \): find the least squares solution of an overdetermined system, that is, solve the least squares problem
   \[
   \minimize \|b - A^Hx\|_2
   \]

Several right hand side vectors \( b \) and solution vectors \( x \) can be handled in a single call; they are formed by the columns of the right hand side matrix \( B \) and the solution matrix \( X \) (when coefficient matrix is \( A \), \( B \) is \( m\)-by-\( nrhs \) and \( X \) is \( n\)-by-\( nrhs \); if the coefficient matrix is \( A^T \) or \( A^H \), \( B \) is \( n\)-by-\( nrhs \) and \( X \) is \( m\)-by-\( nrhs \)).

Input Parameters

- \textit{matrix_layout} specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \textit{trans} must be \('N'\), \('T'\), or \('C'\).
  - If \( \text{trans} = 'N' \), the linear system involves matrix \( A \);
  - If \( \text{trans} = 'T' \), the linear system involves the transposed matrix \( A^T \) (for real flavors only);
  - If \( \text{trans} = 'C' \), the linear system involves the conjugate-transposed matrix \( A^H \) (for complex flavors only).
- \textit{m} the number of rows of the matrix \( A \) (\( m \geq 0 \)).
- \textit{n} the number of columns of the matrix \( A \) (\( n \geq 0 \)).
- \textit{nrhs} the number of right-hand sides; the number of columns in \( B \) (\( nrhs \geq 0 \)).
- \textit{a, b} arrays:
  - \( a \) (size max(1, \( lda\)*\( n \)) for column major layout and max(1, \( lda\)*\( m \)) for row major layout) contains the \( m\)-by-\( n \) matrix \( A \).
  - \( b \) (size max(1, \( ldb\)*\( nrhs \)) for column major layout and max(1, \( ldb\)*max(\( m \), \( n \)) for row major layout) contains the matrix \( B \) of right hand side vectors.
- \textit{lda} the leading dimension of \( a \); at least max(1, \( m \)) for column major layout and at least max(1, \( n \)) for row major layout.
The leading dimension of $b$; must be at least \( \max(1, m, n) \) for column major layout if \( \text{trans} = \text{'N'} \) and at least \( \max(1, n) \) if \( \text{trans} = \text{'T'} \) and at least \( \max(1, \text{nrhs}) \) for row major layout regardless of the value of \( \text{trans} \).

Output Parameters

$a$

On exit, overwritten by the factorization data as follows:
- If \( m \geq n \), array \( a \) contains the details of the QR factorization of the matrix \( A \) as returned by \( \text{gesqr} \);
- If \( m < n \), array \( a \) contains the details of the LQ factorization of the matrix \( A \) as returned by \( \text{gelsq} \).

If \( \text{info} = 0 \), \( b \) overwritten by the solution vectors, stored columnwise:
- If \( \text{trans} = \text{'N'} \) and \( m \geq n \), rows 1 to \( n \) of \( b \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements \( n+1 \) to \( m \) in that column;
- If \( \text{trans} = \text{'N'} \) and \( m < n \), rows 1 to \( n \) of \( b \) contain the minimum norm solution vectors;
- If \( \text{trans} = \text{'T'} \) or \( \text{'C'} \) and \( m \geq n \), rows 1 to \( m \) of \( b \) contain the minimum norm solution vectors;
- If \( \text{trans} = \text{'T'} \) or \( \text{'C'} \) and \( m < n \), rows 1 to \( m \) of \( b \) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of modulus of elements \( m+1 \) to \( n \) in that column.

Return Values

This function returns a value \( \text{info} \).
- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.
- If \( \text{info} = i \), the \( i \)-th diagonal element of the triangular factor of \( A \) is zero, so that \( A \) does not have full rank; the least squares solution could not be computed.

??gelsy

Computes the minimum-norm solution to a linear least squares problem using a complete orthogonal factorization of \( A \).

Syntax

For single precision:

```
lapack_int LAPACKE_sgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, lapack_int* jpvt, float rcond, lapack_int* rank );
```

For double precision:

```
lapack_int LAPACKE_dgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, lapack_int* jpvt, double rcond, lapack_int* rank );
```

For complex single precision:

```
lapack_int LAPACKE_cgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_int* jpvt, float rcond, lapack_int* rank );
```
lapack_int LAPACKE_zgelsy( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_int* jpvt, double rcond, lapack_int* rank );

**Include Files**
- mkl.h

**Description**
The ?gelsy routine computes the minimum-norm solution to a real/complex linear least squares problem:

\[
\text{minimize } \|b - Ax\|_2
\]

using a complete orthogonal factorization of \( A \). \( A \) is an \( m \)-by-\( n \) matrix which may be rank-deficient. Several right hand side vectors \( b \) and solution vectors \( x \) can be handled in a single call; they are stored as the columns of the \( m \)-by-\( nrhs \) right hand side matrix \( B \) and the \( n \)-by-\( nrhs \) solution matrix \( X \).

The routine first computes a \( QR \) factorization with column pivoting:

\[
AP = Q \begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix}
\]

with \( R_{11} \) defined as the largest leading submatrix whose estimated condition number is less than \( 1/rcond \). The order of \( R_{11} \), \( \text{rank} \), is the effective rank of \( A \). Then, \( R_{22} \) is considered to be negligible, and \( R_{12} \) is annihilated by orthogonal/unitary transformations from the right, arriving at the complete orthogonal factorization:

\[
AP = Q \begin{pmatrix} T_{11} & 0 \\ 0 & 0 \end{pmatrix} Z
\]

The minimum-norm solution is then

\[
X = PZ^T \begin{pmatrix} T_{11}^{-1}Q_1^T B \\ 0 \end{pmatrix}
\]

for real flavors and

\[
X = PZ^H \begin{pmatrix} T_{11}^{-1}Q_1^H B \\ 0 \end{pmatrix}
\]

for complex flavors,

where \( Q_1 \) consists of the first \( \text{rank} \) columns of \( Q \).

The ?gelsy routine is identical to the original deprecated ?gelsx routine except for the following differences:
- The call to the subroutine ?geqpf has been substituted by the call to the subroutine ?geqp3, which is a BLAS-3 version of the \( QR \) factorization with column pivoting.
- The matrix \( B \) (the right hand side) is updated with BLAS-3.
- The permutation of the matrix \( B \) (the right hand side) is faster and more simple.

**Input Parameters**

*matrix_layout* Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
The number of rows of the matrix $A$ ($m \geq 0$).

The number of columns of the matrix $A$ ($n \geq 0$).

The number of right-hand sides; the number of columns in $B$ ($nrhs \geq 0$).

Arrays:

- $a$ (size $max(1, lda*n)$ for column major layout and $max(1, lda*m)$ for row major layout) contains the $m$-by-$n$ matrix $A$.
- $b$ (size $max(1, ldb*nrhs)$ for column major layout and $max(1, ldb*max(m, n))$ for row major layout) contains the $m$-by-$nrhs$ right hand side matrix $B$.

The leading dimension of $a$; at least $max(1, m)$ for column major layout and $max(1, n)$ for row major layout.

The leading dimension of $b$; must be at least $max(1, m, n)$ for column major layout and at least $max(1, nrhs)$ for row major layout.

Array, size at least $max(1, n)$.

On entry, if $jpvt[i - 1] \neq 0$, the $i$-th column of $A$ is permuted to the front of $AP$, otherwise the $i$-th column of $A$ is a free column.

$rcond$ is used to determine the effective rank of $A$, which is defined as the order of the largest leading triangular submatrix $R_{11}$ in the $QR$ factorization with pivoting of $A$, whose estimated condition number $< 1/rcond$.

Output Parameters

- $a$ On exit, overwritten by the details of the complete orthogonal factorization of $A$.
- $b$ Overwritten by the $n$-by-$nrhs$ solution matrix $X$.
- $jpvt$ On exit, if $jpvt[i - 1] = k$, then the $i$-th column of $AP$ was the $k$-th column of $A$.
- $rank$ The effective rank of $A$, that is, the order of the submatrix $R_{11}$. This is the same as the order of the submatrix $T_{11}$ in the complete orthogonal factorization of $A$.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

$\texttt{?gelss}$

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of $A$. 
Syntax

```c
lapack_int LAPACKE_sgelss(int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond, lapack_int* rank);

lapack_int LAPACKE_dgelss(int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond, lapack_int* rank);

lapack_int LAPACKE_cgelss(int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float* s, float rcond, lapack_int* rank);

lapack_int LAPACKE_zgelss(int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, double* s, double rcond, lapack_int* rank);
```

Include Files

- mkl.h

Description

The routine computes the minimum norm solution to a real linear least squares problem:

\[
\minimize_{x} ||b - Ax||_2
\]

using the singular value decomposition (SVD) of \(A\). \(A\) is an \(m\)-by-\(n\) matrix which may be rank-deficient. Several right hand side vectors \(b\) and solution vectors \(x\) can be handled in a single call; they are stored as the columns of the \(m\)-by-\(nrhs\) right hand side matrix \(B\) and the \(n\)-by-\(nrhs\) solution matrix \(X\). The effective rank of \(A\) is determined by treating as zero those singular values which are less than \(rcond\) times the largest singular value.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `m` The number of rows of the matrix \(A\) \((m \geq 0)\).
- `n` The number of columns of the matrix \(A\) \((n \geq 0)\).
- `nrhs` The number of right-hand sides; the number of columns in \(B\) \((nrhs \geq 0)\).
- `a`, `b` Arrays:
  - \(a\)(size max(1, \(lda\)*\(n\)) for column major layout and max(1, \(lda\)*\(m\)) for row major layout) contains the \(m\)-by-\(n\) matrix \(A\).
  - \(b\)(size max(1, \(ldb\)*\(nrhs\)) for column major layout and max(1, \(ldb\)*max(\(m\), \(n\))) for row major layout) contains the \(m\)-by-\(nrhs\) right hand side matrix \(B\).
- `lda` The leading dimension of \(a\); at least max(1, \(m\)) for column major layout and max(1, \(n\)) for row major layout.
ldb

The leading dimension of b; must be at least max(1, m, n) for column major layout and at least max(1, nrhs) for row major layout.

rcond

$rcond$ is used to determine the effective rank of $A$. Singular values $s(i) < rcond * s(1)$ are treated as zero.

If $rcond < 0$, machine precision is used instead.

Output Parameters

$a$

On exit, the first min(m, n) rows of $a$ are overwritten with the matrix of right singular vectors of $A$, stored row-wise.

$b$

Overwritten by the $n$-by-$nrhs$ solution matrix $X$.

If $m \geq n$ and $rank = n$, the residual sum-of-squares for the solution in the $i$-th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

$s$

Array, size at least max(1, min(m, n)). The singular values of $A$ in decreasing order. The condition number of $A$ in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.

$rank$

The effective rank of $A$, that is, the number of singular values which are greater than $rcond * s(1)$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, then the algorithm for computing the SVD failed to converge; $i$ indicates the number of off-diagonal elements of an intermediate bidiagonal form which did not converge to zero.

$\texttt{gelsd}$

Computes the minimum-norm solution to a linear least squares problem using the singular value decomposition of $A$ and a divide and conquer method.

Syntax

```c
lapack_int LAPACKE_sgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, float* a, lapack_int lda, float* b, lapack_int ldb, float* s, float rcond, lapack_int* rank );

lapack_int LAPACKE_dgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, double* a, lapack_int lda, double* b, lapack_int ldb, double* s, double rcond, lapack_int* rank );

lapack_int LAPACKE_cgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float* s, float rcond, lapack_int* rank );

lapack_int LAPACKE_zgelsd( int matrix_layout, lapack_int m, lapack_int n, lapack_int nrhs, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, double* s, double rcond, lapack_int* rank );
```
Include Files

- mkl.h

Description

The routine computes the minimum-norm solution to a real linear least squares problem:

$$\min_{x} ||b - A^*x||_2$$

using the singular value decomposition (SVD) of \(A\). \(A\) is an \(m\)-by-\(n\) matrix which may be rank-deficient.

Several right hand side vectors \(b\) and solution vectors \(x\) can be handled in a single call; they are stored as the columns of the \(m\)-by-\(nrhs\) right hand side matrix \(B\) and the \(n\)-by-\(nrhs\) solution matrix \(X\).

The problem is solved in three steps:

1. Reduce the coefficient matrix \(A\) to bidiagonal form with Householder transformations, reducing the original problem into a "bidiagonal least squares problem" (BLS).
2. Solve the BLS using a divide and conquer approach.
3. Apply back all the Householder transformations to solve the original least squares problem.

The effective rank of \(A\) is determined by treating as zero those singular values which are less than \(rcond\) times the largest singular value.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **m**
  The number of rows of the matrix \(A\) \((m \geq 0)\).
- **n**
  The number of columns of the matrix \(A\) \((n \geq 0)\).
- **nrhs**
  The number of right-hand sides; the number of columns in \(B\) \((nrhs \geq 0)\).
- **a, b**
  Arrays:
  - \(a\) (size max(1, \(lda\)*\(n\)) for column major layout and max(1, \(lda\)*\(m\)) for row major layout) contains the \(m\)-by-\(n\) matrix \(A\).
  - \(b\) (size max(1, \(ldb\)*\(nrhs\)) for column major layout and max(1, \(ldb\)*max(\(m\), \(n\))) for row major layout) contains the \(m\)-by-\(nrhs\) right hand side matrix \(B\).
- **lda**
  The leading dimension of \(a\); at least max(1, \(m\)) for column major layout and max(1, \(n\)) for row major layout.
- **ldb**
  The leading dimension of \(b\); must be at least max(1, \(m\), \(n\)) for column major layout and at least max(1, \(nrhs\)) for row major layout.
- **rcond**
  \(rcond\) is used to determine the effective rank of \(A\). Singular values \(s(i) \leq rcond \cdot s(1)\) are treated as zero. If \(rcond \leq 0\), machine precision is used instead.

Output Parameters

- **a**
  On exit, \(A\) has been overwritten.
- **b**
  Overwritten by the \(n\)-by-\(nrhs\) solution matrix \(X\).
If $m \geq n$ and $\text{rank} = n$, the residual sum-of-squares for the solution in the $i$-th column is given by the sum of squares of modulus of elements $n+1:m$ in that column.

$s$

Array, size at least $\max(1, \min(m, n))$. The singular values of $A$ in decreasing order. The condition number of $A$ in the 2-norm is $k_2(A) = s(1) / s(\min(m, n))$.

$\text{rank}$

The effective rank of $A$, that is, the number of singular values which are greater than $rcond \times s(1)$.

**Return Values**

This function returns a value $\text{info}$.

If $\text{info}=0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.

If $\text{info} = i$, then the algorithm for computing the SVD failed to converge; $i$ indicates the number of off-diagonal elements of an intermediate bidiagonal form that did not converge to zero.

**Generalized Linear Least Squares (LLS) Problems: LAPACK Driver Routines**

This section describes LAPACK driver routines used for solving generalized linear least squares problems. Table "Driver Routines for Solving Generalized LLS Problems" lists all such routines.

**Driver Routines for Solving Generalized LLS Problems**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gglse</td>
<td>Solves the linear equality-constrained least squares problem using a generalized RQ factorization.</td>
</tr>
<tr>
<td>gglm</td>
<td>Solves a general Gauss-Markov linear model problem using a generalized QR factorization.</td>
</tr>
</tbody>
</table>

**Syntax**

```c
lapack_int LAPACKE_sgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
float* a, lapack_int lda, float* b, lapack_int ldb, float* c, float* d, float* x);

lapack_int LAPACKE_dgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
double* a, lapack_int lda, double* b, lapack_int ldb, double* c, double* d, double* x);

lapack_int LAPACKE_cgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* c, lapack_complex_float* d, lapack_complex_float* x);

lapack_int LAPACKE_zgglse (int matrix_layout, lapack_int m, lapack_int n, lapack_int p,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* c, lapack_complex_double* d, lapack_complex_double* x);
```

**Include Files**

- mkl.h
Description

The routine solves the linear equality-constrained least squares (LSE) problem:

\[ \min \| c - A^*x \|^2 \quad \text{subject to} \quad B^*x = d \]

where \( A \) is an \( m \)-by-\( n \) matrix, \( B \) is a \( p \)-by-\( n \) matrix, \( c \) is a given \( m \)-vector, and \( d \) is a given \( p \)-vector. It is assumed that \( p \leq n \leq m + p \), and

\[ \text{rank}(B) = p \quad \text{and} \quad \text{rank} \left( \begin{bmatrix} A \\ B \end{bmatrix} \right) = n. \]

These conditions ensure that the LSE problem has a unique solution, which is obtained using a generalized RQ factorization of the matrices \((B, A)\) given by

\[ B = (0 \quad R)^*Q, \quad A = Z^*T^*Q \]

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).
- **m**: The number of rows of the matrix \( A \) \((m \geq 0)\).
- **n**: The number of columns of the matrices \( A \) and \( B \) \((n \geq 0)\).
- **p**: The number of rows of the matrix \( B \)
  \((0 \leq p \leq n \leq m + p)\).
- **a**, **b**, **c**, **d**: Arrays:
  - \( a \) (size \( \text{max}(1, lda*n) \) for column major layout and \( \text{max}(1, lda*m) \) for row major layout) contains the \( m \)-by-\( n \) matrix \( A \).
  - \( b \) (size \( \text{max}(1, ldb*n) \) for column major layout and \( \text{max}(1, ldb*p) \) for row major layout) contains the \( p \)-by-\( n \) matrix \( B \).
  - \( c \) size at least \( \text{max}(1, m) \), contains the right hand side vector for the least squares part of the LSE problem.
  - \( d \), size at least \( \text{max}(1, p) \), contains the right hand side vector for the constrained equation.
- **lda**: The leading dimension of \( a \); at least \( \text{max}(1, m) \) for column major layout and \( \text{max}(1, n) \) for row major layout.
- **ldb**: The leading dimension of \( b \); at least \( \text{max}(1, p) \) for column major layout and \( \text{max}(1, n) \) for row major layout.

Output Parameters

- **a**: The elements on and above the diagonal contain the \( \text{min}(m, n) \)-by-\( n \) upper trapezoidal matrix \( T \) as returned by \( ?ggrqf \).
- **x**: The solution of the LSE problem.
- **b**: On exit, the upper right triangle contains the \( p \)-by-\( p \) upper triangular matrix \( R \) as returned by \( ?ggrqf \).
On exit, \( d \) is destroyed.

On exit, the residual sum-of-squares for the solution is given by the sum of squares of elements \( n-p+1 \) to \( m \) of vector \( c \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = 1 \), the upper triangular factor \( R \) associated with \( B \) in the generalized RQ factorization of the pair \((B, A)\) is singular, so that \( \text{rank}(B) < p \); the least squares solution could not be computed.

If \( info = 2 \), the \((n-p)\)-by-\((n-p)\) part of the upper trapezoidal factor \( T \) associated with \( A \) in the generalized RQ factorization of the pair \((B, A)\) is singular, so that 

\[
\text{rank} \begin{pmatrix} A \\ B \end{pmatrix} < n
\]

the least squares solution could not be computed.

**?ggglm**

Solves a general Gauss-Markov linear model problem using a generalized QR factorization.

**Syntax**

```c
lapack_int LAPACKE_sggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
float* a, lapack_int lda, float* b, lapack_int ldb, float* d, float* x, float* y);

lapack_int LAPACKE_dggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
double* a, lapack_int lda, double* b, lapack_int ldb, double* d, double* x, double* y);

lapack_int LAPACKE_cggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
lapack_complex_float* d, lapack_complex_float* x, lapack_complex_float* y);

lapack_int LAPACKE_zggglm (int matrix_layout, lapack_int n, lapack_int m, lapack_int p,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
lapack_complex_double* d, lapack_complex_double* x, lapack_complex_double* y);
```

**Include Files**

- mkl.h

**Description**

The routine solves a general Gauss-Markov linear model (GLM) problem:

\[
\text{minimize}_{x} ||y||_2 \text{ subject to } d = A*x + B*y
\]

where \( A \) is an \( n \)-by-\( m \) matrix, \( B \) is an \( n \)-by-\( p \) matrix, and \( d \) is a given \( n \)-vector. It is assumed that \( m \leq n \leq m + p \), and \( \text{rank}(A) = m \) and \( \text{rank}(AB) = n \).

Under these assumptions, the constrained equation is always consistent, and there is a unique solution \( x \) and a minimal 2-norm solution \( y \), which is obtained using a generalized QR factorization of the matrices \((A, B)\) given by
In particular, if matrix $B$ is square nonsingular, then the problem GLM is equivalent to the following weighted linear least squares problem

$$\minimize_x ||B^{-1}(d-A^*x)||_2.$$ 

**Input Parameters**

$matrix\_layout$  
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

$n$  
The number of rows of the matrices $A$ and $B$ ($n \geq 0$).

$m$  
The number of columns in $A$ ($m \geq 0$).

$p$  
The number of columns in $B$ ($p \geq n - m$).

$a$, $b$, $d$  
Arrays:

- $a$ (size max(1, $lda*m$) for column major layout and max(1, $lda*n$) for row major layout) contains the $n$-by-$m$ matrix $A$.
- $b$ (size max(1, $ldb*p$) for column major layout and max(1, $ldb*n$) for row major layout) contains the $n$-by-$p$ matrix $B$.
- $d$, size at least max(1, $n$), contains the left hand side of the GLM equation.

$lda$  
The leading dimension of $a$; at least max(1, $n$) for column major layout and max(1, $m$) for row major layout.

$ldb$  
The leading dimension of $b$; at least max(1, $n$) for column major layout and max(1, $p$) for row major layout.

**Output Parameters**

$x$, $y$  
Arrays $x$, $y$. size at least max(1, $m$) for $x$ and at least max(1, $p$) for $y$.

On exit, $x$ and $y$ are the solutions of the GLM problem.

$a$  
On exit, the upper triangular part of the array $a$ contains the $m$-by-$m$ upper triangular matrix $R$.

$b$  
On exit, if $n \leq p$, the upper right triangle contains the $n$-by-$n$ upper triangular matrix $T$ as returned by ?ggrqf; if $n > p$, the elements on and above the ($n-p$)-th subdiagonal contain the $n$-by-$p$ upper trapezoidal matrix $T$.

$d$  
On exit, $d$ is destroyed

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.
If $info = 1$, the upper triangular factor $R$ associated with $A$ in the generalized QR factorization of the pair $(A, B)$ is singular, so that $\text{rank}(A) < m$; the least squares solution could not be computed.

If $info = 2$, the bottom $(n-m)$-by-$(n-m)$ part of the upper trapezoidal factor $T$ associated with $B$ in the generalized QR factorization of the pair $(A, B)$ is singular, so that $\text{rank}(AB) < n$; the least squares solution could not be computed.

**Symmetric Eigenvalue Problems: LAPACK Driver Routines**

This section describes LAPACK driver routines used for solving symmetric eigenvalue problems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Symmetric Eigenproblems" lists all such driver routines.

**Driver Routines for Solving Symmetric Eigenproblems**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>syev/heev</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix.</td>
</tr>
<tr>
<td>syevd/heevd</td>
<td>Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix using divide and conquer algorithm.</td>
</tr>
<tr>
<td>syevx/heevx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a symmetric / Hermitian matrix.</td>
</tr>
<tr>
<td>syevr/heevr</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix using the Relatively Robust Representations.</td>
</tr>
<tr>
<td>spev/hpev</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.</td>
</tr>
<tr>
<td>spevd/hpevd</td>
<td>Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian matrix held in packed storage.</td>
</tr>
<tr>
<td>spevx/hpevx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian matrix in packed storage.</td>
</tr>
<tr>
<td>sbev/hbev</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.</td>
</tr>
<tr>
<td>sbevd/hbevd</td>
<td>Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric / Hermitian band matrix using divide and conquer algorithm.</td>
</tr>
<tr>
<td>sbevx/hbevx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric / Hermitian band matrix.</td>
</tr>
<tr>
<td>stev</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.</td>
</tr>
<tr>
<td>stevd</td>
<td>Computes all eigenvalues and (optionally) all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.</td>
</tr>
<tr>
<td>stevx</td>
<td>Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.</td>
</tr>
<tr>
<td>stevr</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.</td>
</tr>
</tbody>
</table>

syev

*Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix.*
Syntax

lapack_int LAPACKE_ssyev (int matrix_layout, char jobz, char uplo, lapack_int n, float* a, lapack_int lda, float* w);

lapack_int LAPACKE_dsyev (int matrix_layout, char jobz, char uplo, lapack_int n, double* a, lapack_int lda, double* w);

Include Files

• mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix A. Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function as its underlying algorithm is faster and uses less workspace.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz

Must be 'N' or 'V'.

If jobz = 'N', then only eigenvalues are computed.

If jobz = 'V', then eigenvalues and eigenvectors are computed.

uplo

Must be 'U' or 'L'.

If uplo = 'U', a stores the upper triangular part of A.

If uplo = 'L', a stores the lower triangular part of A.

n

The order of the matrix A (n ≥ 0).

a

a (size max(1, lda*n)) is an array containing either upper or lower triangular part of the symmetric matrix A, as specified by uplo.

lda

The leading dimension of the array a.

Must be at least max(1, n).

Output Parameters

a

On exit, if jobz = 'V', then if info = 0, array a contains the orthonormal eigenvectors of the matrix A.

If jobz = 'N', then on exit the lower triangle (if uplo = 'L') or the upper triangle (if uplo = 'U') of A, including the diagonal, is overwritten.

w

Array, size at least max(1, n).

If info = 0, contains the eigenvalues of the matrix A in ascending order.

Return Values

This function returns a value info.
If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info = i, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

**heev**

*Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix.*

**Syntax**

lapack_int LAPACKE_cheev( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, float* w);

lapack_int LAPACKE_zheev( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda, double* w);

**Include Files**

- mkl.h

**Description**

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be heevr function as its underlying algorithm is faster and uses less workspace.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  
  Must be 'N' or 'V'.
  
  If jobz = 'N', then only eigenvalues are computed.
  
  If jobz = 'V', then eigenvalues and eigenvectors are computed.

- **uplo**
  
  Must be 'U' or 'L'.
  
  If uplo = 'U', a stores the upper triangular part of A.
  
  If uplo = 'L', a stores the lower triangular part of A.

- **n**
  
  The order of the matrix A (n ≥ 0).

- **a**
  
  a (size max(1, lda*n)) is an array containing either upper or lower triangular part of the Hermitian matrix A, as specified by uplo.

- **lda**
  
  The leading dimension of the array a. Must be at least max(1, n).

**Output Parameters**

- **a**
  
  On exit, if jobz = 'V', then if info = 0, array a contains the orthonormal eigenvectors of the matrix A.
  
  If jobz = 'N', then on exit the lower triangle
(if uplo = 'L') or the upper triangle (if uplo = 'U') of A, including the
diagonal, is overwritten.

w
Array, size at least max(1, n).
If info = 0, contains the eigenvalues of the matrix A in ascending order.

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
If info = i, then the algorithm failed to converge; i indicates the number of elements of an intermediate
tridiagonal form which did not converge to zero.

?syevd
Computes all eigenvalues and, optionally, all
eigenvectors of a real symmetric matrix using divide
and conquer algorithm.

Syntax

lapack_int LAPACKE_ssyevd (int matrix_layout, char jobz, char uplo, lapack_int n,
float* a, lapack_int lda, float* w);

lapack_int LAPACKE_dsyevd (int matrix_layout, char jobz, char uplo, lapack_int n,
double* a, lapack_int lda, double* w);

Include Files
• mkl.h

Description
The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix A.
In other words, it can compute the spectral factorization of A as: A = Z*Λ*Z^T.

Here Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix
whose columns are the eigenvectors z_i. Thus,
A*z_i = λ_i*z_i for i = 1, 2, ..., n.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute
eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan
variant of the QL or QR algorithm.

Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function
as its underlying algorithm is faster and uses less workspace. ?syevd requires more workspace but is faster
in some cases, especially for large matrices.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
or column major (LAPACK_COL_MAJOR).

jobz
Must be 'N' or 'V'.
If jobz = 'N', then only eigenvalues are computed.
If \( jobz = 'V' \), then eigenvalues and eigenvectors are computed.

**uplo**

Must be 'U' or 'L'.

If \( uplo = 'U' \), \( a \) stores the upper triangular part of \( A \).

If \( uplo = 'L' \), \( a \) stores the lower triangular part of \( A \).

**n**

The order of the matrix \( A \) (\( n \geq 0 \)).

**a**

Array, size (\( lda, * \)).

\( a \) (size max(1, \( lda*n \))) is an array containing either upper or lower triangular part of the symmetric matrix \( A \), as specified by \( uplo \).

**lda**

The leading dimension of the array \( a \).

Must be at least max(1, \( n \)).

**Output Parameters**

**w**

Array, size at least max(1, \( n \)).

If \( info = 0 \), contains the eigenvalues of the matrix \( A \) in ascending order. See also \( info \).

**a**

If \( jobz = 'V' \), then on exit this array is overwritten by the orthogonal matrix \( Z \) which contains the eigenvectors of \( A \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = i, and \ jobz = 'N' \), then the algorithm failed to converge; \( i \) indicates the number of off-diagonal elements of an intermediate tridiagonal form which did not converge to zero.

If \( info = i, and \ jobz = 'V' \), then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns \( info/(n+1) \) through \( mod(info,n+1) \).

If \( info = -i \), the \( i \)-th parameter had an illegal value.

**Application Notes**

The computed eigenvalues and eigenvectors are exact for a matrix \( A+E \) such that \( ||E||_2 = O(\varepsilon) * ||A||_2 \), where \( \varepsilon \) is the machine precision.

The complex analogue of this routine is heevd?

heevd

Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix using divide and conquer algorithm.

**Syntax**

```c
lapack_int LAPACKE_cheevd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, float* w );

lapack_int LAPACKE_zheevd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda, double* w );
```
Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix $A$. In other words, it can compute the spectral factorization of $A$ as: $A = Z^* \Lambda Z^H$.

Here $\Lambda$ is a real diagonal matrix whose diagonal elements are the eigenvalues $\lambda_i$, and $Z$ is the (complex) unitary matrix whose columns are the eigenvectors $z_i$. Thus,

$$A^* z_i = \lambda_i z_i$$

for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the $QL$ or $QR$ algorithm.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be $heevr$ function as its underlying algorithm is faster and uses less workspace. $heevr$ requires more workspace but is faster in some cases, especially for large matrices.

Input Parameters

$matrix\_layout$  
Specifies whether matrix storage layout is row major ($LAPACK\_ROW\_MAJOR$) or column major ($LAPACK\_COL\_MAJOR$).

$jobz$  
Must be 'N' or 'V'.

If $jobz = 'N'$, then only eigenvalues are computed.

If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

$uplo$  
Must be 'U' or 'L'.

If $uplo = 'U'$, $a$ stores the upper triangular part of $A$.

If $uplo = 'L'$, $a$ stores the lower triangular part of $A$.

$n$  
The order of the matrix $A$ ($n \geq 0$).

$a$  
$a$ (size $\text{max}(1, lda \times n)$) is an array containing either upper or lower triangular part of the Hermitian matrix $A$, as specified by $uplo$.

$lda$  
The leading dimension of the array $a$. Must be at least $\text{max}(1, n)$.

Output Parameters

$w$  
Array, size at least $\text{max}(1, n)$.

If $info = 0$, contains the eigenvalues of the matrix $A$ in ascending order. See also $info$.

$a$  
If $jobz = 'V'$, then on exit this array is overwritten by the unitary matrix $Z$ which contains the eigenvectors of $A$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.
If $info = i$, and $jobz = 'N'$, then the algorithm failed to converge; $i$ off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if $info = i$, and $jobz = 'V'$, then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns $info/(n+1)$ through $\text{mod}(info, n+1)$.

If $info = -i$, the $i$-th parameter had an illegal value.

**Application Notes**

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(e) ||A||_2$, where $e$ is the machine precision.

The real analogue of this routine is syevd. See also hpevd for matrices held in packed storage, and hbevd for banded matrices.

**?syevx**

*Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.*

**Syntax**

```c
lapack_int LAPACKE_ssyevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, float* a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int
iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);
```

```c
lapack_int LAPACKE_dsyevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, double* a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int
iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

**Include Files**

- mkl.h

**Description**

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix $A$. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of real symmetric eigenvalue problems the default choice should be syevr function as its underlying algorithm is faster and uses less workspace. ?syevx is faster for a few selected eigenvalues.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.
  If $jobz = 'N'$, then only eigenvalues are computed.
  If $jobz = 'V'$, then eigenvalues and eigenvectors are computed.

- **range**
  Must be 'A', 'V', or 'I'.
  If $range = 'A'$, all eigenvalues will be found.
If \( \text{range} = 'V' \), all eigenvalues in the half-open interval \((vl, vu]\) will be found.

If \( \text{range} = 'I' \), the eigenvalues with indices \( il \) through \( iu \) will be found.

**uplo**

Must be 'U' or 'L'.

If \( uplo = 'U' \), \( a \) stores the upper triangular part of \( A \).

If \( uplo = 'L' \), \( a \) stores the lower triangular part of \( A \).

**n**

The order of the matrix \( A \) (\( n \geq 0 \)).

\( a \) (size \( \text{max}(1, \text{lda} \times n) \)) is an array containing either upper or lower triangular part of the symmetric matrix \( A \), as specified by \( uplo \).

**lda**

The leading dimension of the array \( a \). Must be at least \( \text{max}(1, n) \).

**vl, vu**

If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues; \( vl \leq vu \). Not referenced if \( \text{range} = 'A' \) or 'I'.

**il, iu**

If \( \text{range} = 'I' \), the indices of the smallest and largest eigenvalues to be returned.

Constraints: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \);

\( il = 1 \) and \( iu = 0 \), if \( n = 0 \).

Not referenced if \( \text{range} = 'A' \) or 'V'.

**abstol**

The absolute error tolerance for the eigenvalues. See Application Notes for more information.

**ldz**

The leading dimension of the output array \( z \); \( ldz \geq 1 \).

If \( \text{jobz} = 'V' \), then \( ldz \geq \text{max}(1, n) \) for column major layout and \( ldz \geq \text{max}(1, m) \) for row major layout.

### Output Parameters

**a**

On exit, the lower triangle (if \( uplo = 'L' \)) or the upper triangle (if \( uplo = 'U' \)) of \( A \), including the diagonal, is overwritten.

The total number of eigenvalues found;

\( 0 \leq m \leq n \).

If \( \text{range} = 'A', m = n \), and if \( \text{range} = 'I', m = iu-il+1 \).

**w**

Array, size at least \( \text{max}(1, n) \). The first \( m \) elements contain the selected eigenvalues of the matrix \( A \) in ascending order.

**z**

Array \( z \)(size \( \text{max}(1, ldz \times m) \) for column major layout and \( \text{max}(1, ldz \times n) \) for row major layout) contains eigenvectors.

If \( \text{jobz} = 'V' \), then if \( info = 0 \), the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w(i) \).
If an eigenvector fails to converge, then that column of $z$ contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.

If `jobz` = 'N', then $z$ is not referenced.

Note: you must ensure that at least max(1, $m$) columns are supplied in the array $z$; if `range` = 'V', the exact value of $m$ is not known in advance and an upper bound must be used.

`ifail` Array, size at least max(1, $n$).

If `jobz` = 'V', then if `info` = 0, the first $m$ elements of `ifail` are zero; if `info` > 0, then `ifail` contains the indices of the eigenvectors that failed to converge.

If `jobz` = 'V', then `ifail` is not referenced.

Return Values

This function returns a value `info`.

If `info`=0, the execution is successful.

If `info` = -i, the i-th parameter had an illegal value.

If `info` = i, then i eigenvectors failed to converge; their indices are stored in the array `ifail`.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \epsilon \max(|a|,|b|)$, where $\epsilon$ is the machine precision.

If $\text{abstol}$ is less than or equal to zero, then $\epsilon \cdot ||T||$ is used as tolerance, where $||T||$ is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues are computed most accurately when $\text{abstol}$ is set to twice the underflow threshold $2\cdot\text{lamch}(\text{\'S}')$, not zero.

If this routine returns with `info` > 0, indicating that some eigenvectors did not converge, try setting $\text{abstol}$ to $2\cdot\text{lamch}(\text{\'S}')$.

`heevx`

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

Syntax

```c
lapack_int LAPACKE_cheevx( int matrix_layout, char jobz, char range, char uplo,
                      lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int il,
                      lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z,
                      lapack_int ldz, lapack_int* ifail );

lapack_int LAPACKE_zheevx( int matrix_layout, char jobz, char range, char uplo,
                      lapack_int n, lapack_complex_double* a, lapack_int lda, double vl, double vu,
                      lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
                      lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h
Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix $A$. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Note that for most cases of complex Hermitian eigenvalue problems the default choice should be `heevr` function as its underlying algorithm is faster and uses less workspace. `?heevx` is faster for a few selected eigenvalues.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**: Must be 'N' or 'V'.
  
  If `jobz` = 'N', then only eigenvalues are computed.
  
  If `jobz` = 'V', then eigenvalues and eigenvectors are computed.

- **range**: Must be 'A', 'V', or 'I'.
  
  If `range` = 'A', all eigenvalues will be found.
  
  If `range` = 'V', all eigenvalues in the half-open interval $(vl, vu]$ will be found.
  
  If `range` = 'I', the eigenvalues with indices $il$ through $iu$ will be found.

- **uplo**: Must be 'U' or 'L'.
  
  If `uplo` = 'U', $a$ stores the upper triangular part of $A$.
  
  If `uplo` = 'L', $a$ stores the lower triangular part of $A$.

- **n**: The order of the matrix $A$ ($n \geq 0$).

- **a**: ($\text{size max}(1, \text{lda} \times n)$) is an array containing either upper or lower triangular part of the Hermitian matrix $A$, as specified by `uplo`.

- **lda**: The leading dimension of the array $a$. Must be at least $\text{max}(1, n)$.

- **vl, vu**: If `range` = 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if `range` = 'A' or 'I'.

- **il, iu**: If `range` = 'I', the indices of the smallest and largest eigenvalues to be returned. Constraints:
  
  $1 \leq il \leq n$, if $n > 0$; $il = 1$ and $iu = 0$, if $n = 0$. Not referenced if `range` = 'A' or 'V'.

- **abstol**: The leading dimension of the output array $z$; $ldz \geq 1$.

- **ldz**: The leading dimension of the output array $z$; $ldz \geq 1$ for column major layout and $lda \geq \text{max}(1, m)$ for row major layout.
Output Parameters

- **a**: On exit, the lower triangle (if uplo = 'L') or the upper triangle (if uplo = 'U') of A, including the diagonal, is overwritten.

- **m**: The total number of eigenvalues found; 0 ≤ m ≤ n.
  If range = 'A', m = n, and if range = 'I', m = il - ii + 1.

- **w**: Array, size max(1, n). The first m elements contain the selected eigenvalues of the matrix A in ascending order.

- **z**: Array z(size max(1, ldz*m) for column major layout and max(1, ldz*n) for row major layout) contains eigenvectors.
  If jobz = 'V', then if info = 0, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with w(i).
  If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.
  If jobz = 'N', then z is not referenced.

- **ifail**: Array, size at least max(1, n).
  If jobz = 'V', then if info = 0, the first m elements of ifail are zero; if info > 0, then ifail contains the indices of the eigenvectors that failed to converge.
  If jobz = 'V', then ifail is not referenced.

Return Values

This function returns a value info.

- If info=0, the execution is successful.
- If info = -i, the i-th parameter had an illegal value.
- If info = i, then i eigenvectors failed to converge; their indices are stored in the array ifail.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to abstol+ε*max(|a|,|b|), where ε is the machine precision.

If abstol is less than or equal to zero, then ε*||T|| will be used in its place, where ||T|| is the 1-norm of the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when abstol is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with info > 0, indicating that some eigenvectors did not converge, try setting abstol to 2*?lamch('S').

?syevr
Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using the Relatively Robust Representations.
Syntax

```c
lapack_int LAPACKE_ssyevr (int matrix_layout, char jobz, char range, char uplo,
            lapack_int n, float* a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int iu,
            float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz);

lapack_int LAPACKE_dsyevr (int matrix_layout, char jobz, char range, char uplo,
            lapack_int n, double* a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int iu,
            double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix \(A\). Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix \(A\) to tridiagonal form \(T\). Then, whenever possible, \(?syevr\) calls \(stemr\) to compute the eigenspectrum using Relatively Robust Representations. \(stemr\) computes eigenvalues by the \(dqds\) algorithm, while orthogonal eigenvectors are computed from various “good” \(L*D*L^T\) representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the each unreduced block of \(T\):

- \(a\). Compute \(T - \sigma I = L*D*L^T\), so that \(L\) and \(D\) define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of \(D\) and \(L\) cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix \(T\) does not have this property in general.
- \(b\). Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).
- \(c\). For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.
- \(d\). For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter \(abstol\).

The routine \(?syevr\) calls \(stemr\) when the full spectrum is requested on machines that conform to the IEEE-754 floating point standard. \(?syevr\) calls \(stebz\) and \(stein\) on non-IEEE machines and when partial spectrum requests are made.

Normal execution of \(?dsyevr\) may create NaNs and infinities and may abort due to a floating point exception in environments that do not handle NaNs and infinities in the IEEE standard default manner.

Note that \(?syevr\) is preferable for most cases of real symmetric eigenvalue problems as its underlying algorithm is fast and uses less workspace.

Input Parameters

- \(matrix_layout\) Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \(jobz\) Must be ‘N’ or ‘V’.
If \( \text{jobz} = 'N' \), then only eigenvalues are computed.
If \( \text{jobz} = 'V' \), then eigenvalues and eigenvectors are computed.

**range**
Must be 'A' or 'V' or 'I'.
If \( \text{range} = 'A' \), the routine computes all eigenvalues.
If \( \text{range} = 'V' \), the routine computes eigenvalues \( w[i] \) in the half-open interval:
\( vl < w[i] \leq vu \).
If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).
For \( \text{range} = 'V' \) or 'I' and \( iu-il < n-1 \), \text{sstebz/dstebz} and \text{sstein/dstein} are called.

**uplo**
Must be 'U' or 'L'.
If \( \text{uplo} = 'U' \), \( a \) stores the upper triangular part of \( A \).
If \( \text{uplo} = 'L' \), \( a \) stores the lower triangular part of \( A \).

**n**
The order of the matrix \( A \) (\( n \geq 0 \)).

**a**
\( a \) (size max(1, \( lda* n \))) is an array containing either upper or lower triangular part of the symmetric matrix \( A \), as specified by \( \text{uplo} \).

**lda**
The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

**vl, vu**
If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: \( vl < vu \).
If \( \text{range} = 'A' \) or 'I', \( vl \) and \( vu \) are not referenced.

**il, iu**
If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint:
\( 1 \leq il \leq iu \leq n \), if \( n > 0 \);
\( il=1 \) and \( iu=0 \), if \( n = 0 \).
If \( \text{range} = 'A' \) or 'V', \( il \) and \( iu \) are not referenced.

**abstol**
If \( \text{jobz} = 'V' \), the eigenvalues and eigenvectors output have residual norms bounded by \( \text{abstol} \), and the dot products between different eigenvectors are bounded by \( \text{abstol} \).
If \( \text{abstol} < n * \text{eps} * ||T|| \), then \( n * \text{eps} * ||T|| \) is used instead, where \( \text{eps} \) is the machine precision, and \( ||T|| \) is the 1-norm of the matrix \( T \). The eigenvalues are computed to an accuracy of \( \text{eps} * ||T|| \) irrespective of \( \text{abstol} \).
If high relative accuracy is important, set \( \text{abstol} \) to ?lamch('S').

**ldz**
The leading dimension of the output array \( z \).
Constraints:
\( ldz \geq 1 \) if \( \text{jobz} = 'N' \) and
\[ ldz \geq \max(1, n) \] for column major layout and \[ ldz \geq \max(1, m) \] for row major layout if \( jobz = 'V' \).

**Output Parameters**

\( a \)

On exit, the lower triangle (if \( uplo = 'L' \)) or the upper triangle (if \( uplo = 'U' \)) of \( A \), including the diagonal, is overwritten.

\( m \)

The total number of eigenvalues found, \( 0 \leq m \leq n \).

If \( range = 'A' \), \( m = n \), if \( range = 'I' \), \( m = iu - il + 1 \), and if \( range = 'V' \) the exact value of \( m \) is not known in advance.

\( w, z \)

Arrays:

- \( w \), size at least \( \max(1, n) \), contains the selected eigenvalues in ascending order, stored in \( w[0] \) to \( w[m-1] \);
- \( z \) (size \( \max(1, ldz \times m) \) for column major layout and \( \max(1, ldz \times n) \) for row major layout).

If \( jobz = 'V' \), then if \( info = 0 \), the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \).

If \( jobz = 'N' \), then \( z \) is not referenced.

\( isuppz \)

Array, size at least \( 2 \times \max(1, m) \).

The support of the eigenvectors in \( z \), i.e., the indices indicating the nonzero elements in \( z \). The \( i \)-th eigenvector is nonzero only in elements \( isuppz[2i - 2] \) through \( isuppz[2i - 1] \). Referenced only if eigenvectors are needed (\( jobz = 'V' \)) and all eigenvalues are needed, that is, \( range = 'A' \) or \( range = 'I' \) and \( il = 1 \) and \( iu = n \).

**Return Values**

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = i \), an internal error has occurred.

**Application Notes**

\texttt{heevr}

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using the Relatively Robust Representations.*

**Syntax**

```c
lapack_int LAPACKE_cheevr( int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* isuppz );
```
The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix $A$. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

The routine first reduces the matrix $A$ to tridiagonal form $T$ with a call to `hetrd`. Then, whenever possible, `heevr` calls `stegr` to compute the eigenspectrum using Relatively Robust Representations. `stegr` computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various "good" $L*D*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For each unreduced block (submatrix) of $T$:

a. Compute $T - \sigma I = L*D*L^T$, so that $L$ and $D$ define all the wanted eigenvalues to high relative accuracy. This means that small relative changes in the entries of $D$ and $L$ cause only small relative changes in the eigenvalues and eigenvectors. The standard (unfactored) representation of the tridiagonal matrix $T$ does not have this property in general.

b. Compute the eigenvalues to suitable accuracy. If the eigenvectors are desired, the algorithm attains full accuracy of the computed eigenvalues only right before the corresponding vectors have to be computed, see Steps c) and d).

c. For each cluster of close eigenvalues, select a new shift close to the cluster, find a new factorization, and refine the shifted eigenvalues to suitable accuracy.

d. For each eigenvalue with a large enough relative separation, compute the corresponding eigenvector by forming a rank revealing twisted factorization. Go back to Step c) for any clusters that remain.

The desired accuracy of the output can be specified by the input parameter `abstol`.

The routine `heevr` calls `stemr` when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard, or `stebz` and `stein` on non-IEEE machines and when partial spectrum requests are made.

Note that the routine `heevr` is preferable for most cases of complex Hermitian eigenvalue problems as its underlying algorithm is fast and uses less workspace.

### Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).
- **jobz**: Must be `'N'` or `'V'`. If `jobz = 'N'`, then only eigenvalues are computed. If `jobz = 'V'`, then eigenvalues and eigenvectors are computed.
- **range**: Must be `'A'` or `'V'` or `'I'`. If `range = 'A'`, the routine computes all eigenvalues. If `range = 'V'`, the routine computes eigenvalues $\lambda(i)$ in the half-open interval: $vl < \lambda(i) \leq vu$. 

Include Files

- `mkl.h`
If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).

For \( \text{range} = 'V' \) or \( 'I' \), \text{sstebz/dstebz} and \text{cstein/zstein} are called.

**uplo**

Must be \( 'U' \) or \( 'L' \).

If \( \text{uplo} = 'U' \), \( a \) stores the upper triangular part of \( A \).

If \( \text{uplo} = 'L' \), \( a \) stores the lower triangular part of \( A \).

**n**

The order of the matrix \( A \) (\( n \geq 0 \)).

**a**

\( a \) (size max(1, \( lda \times n \))) is an array containing either upper or lower triangular part of the Hermitian matrix \( A \), as specified by \( \text{uplo} \).

**lda**

The leading dimension of the array \( a \).

Must be at least max(1, \( n \)).

**vl, vu**

If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If \( \text{range} = 'A' \) or \( 'I' \), \( vl \) and \( vu \) are not referenced.

**il, iu**

If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il=1 \) and \( iu=0 \) if \( n = 0 \).

If \( \text{range} = 'A' \) or \( 'V' \), \( il \) and \( iu \) are not referenced.

**abstol**

The absolute error tolerance to which each eigenvalue/eigenvector is required.

If \( \text{jobz} = 'V' \), the eigenvalues and eigenvectors output have residual norms bounded by \( \text{abstol} \), and the dot products between different eigenvectors are bounded by \( \text{abstol} \).

If \( \text{abstol} < n \times \text{eps} \times ||T|| \), then \( n \times \text{eps} \times ||T|| \) is used instead, where \( \text{eps} \) is the machine precision, and \( ||T|| \) is the 1-norm of the matrix \( T \). The eigenvalues are computed to an accuracy of \( \text{eps} \times ||T|| \) irrespective of \( \text{abstol} \).

If high relative accuracy is important, set \( \text{abstol} \) to \( ?\text{lamch}('S') \).

**ldz**

The leading dimension of the output array \( z \). Constraints:

\( ldz \geq 1 \) if \( \text{jobz} = 'N' \);

\( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout if \( \text{jobz} = 'V' \).

**Output Parameters**

\( a \)

On exit, the lower triangle (if \( \text{uplo} = 'L' \)) or the upper triangle (if \( \text{uplo} = 'U' \)) of \( A \), including the diagonal, is overwritten.

\( m \)

The total number of eigenvalues found,

\( 0 \leq m \leq n \).
If range = 'A', m = n, if range = 'I', m = il-iu+1, and if range = 'V' the exact value of m is not known in advance.

w

Array, size at least max(1, n), contains the selected eigenvalues in ascending order, stored in w[0] to w[m-1].

z

Array z(size max(1, ldz*m) for column major layout and max(1, ldz*n) for row major layout).

If jobz = 'V', then if info = 0, the first m columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the i-th column of z holding the eigenvector associated with w[i-1].

If jobz = 'N', then z is not referenced.

isuppz

Array, size at least 2 *max(1, m).

The support of the eigenvectors in z, i.e., the indices indicating the nonzero elements in z. The i-th eigenvector is nonzero only in elements isuppz[2i-2] through isuppz[2i-1]. Referenced only if eigenvectors are needed (jobz = 'V') and all eigenvalues are needed, that is, range = 'A' or range = 'I' and il = 1 and iu = n.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info = i, an internal error has occurred.

Application Notes

Normal execution of ?stemr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

For more details, see ?stemr and these references:


?spev
Computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.

Syntax

```c
lapack_int LAPACKE_sspev (int matrix_layout, char jobz, char uplo, lapack_int n, float* ap, float* w, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dspev (int matrix_layout, char jobz, char uplo, lapack_int n, double* ap, double* w, double* z, lapack_int ldz);
```
Include Files

- mkl.h

Description

The routine computes all the eigenvalues and, optionally, eigenvectors of a real symmetric matrix \( A \) in packed storage.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.
  - If job = 'N', then only eigenvalues are computed.
  - If job = 'V', then eigenvalues and eigenvectors are computed.

- **uplo**
  Must be 'U' or 'L'.
  - If uplo = 'U', \( ap \) stores the packed upper triangular part of \( A \).
  - If uplo = 'L', \( ap \) stores the packed lower triangular part of \( A \).

- **n**
  The order of the matrix \( A \) (\( n \geq 0 \)).

- **ap**
  Array \( ap \) contains the packed upper or lower triangle of symmetric matrix \( A \), as specified by \( uplo \).
  The size of \( ap \) must be at least \( \max(1, n*(n+1)/2) \).

- **ldz**
  The leading dimension of the output array \( z \). Constraints:
  - if jobz = 'N', then \( ldz \geq 1 \);
  - if jobz = 'V', then \( ldz \geq \max(1, n) \).

Output Parameters

- **w, z**
  Arrays:
  - \( w \), size at least \( \max(1, n) \).
    - If info = 0, \( w \) contains the eigenvalues of the matrix \( A \) in ascending order.
    - \( z \) (size \( \max(1, ldz*n) \)).
    - If jobz = 'V', then if info = 0, \( z \) contains the orthonormal eigenvectors of the matrix \( A \), with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i - 1] \).
    - If jobz = 'N', then \( z \) is not referenced.

- **ap**
  On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of \( A \).

Return Values

This function returns a value info.
If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; $i$ indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

$hpev$

*Computes all eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.*

**Syntax**

```c
lapack_int LAPACKE_chpev( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );
```

```c
lapack_int LAPACKE_zhpev( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix $A$ in packed storage.

**Input Parameters**

- $matrix_layout$
  
  Specifies whether matrix storage layout is row major ($\text{LAPACK\_ROW\_MAJOR}$) or column major ($\text{LAPACK\_COL\_MAJOR}$).

- $jobz$
  
  Must be 'N' or 'V'.

  If $job = 'N'$, then only eigenvalues are computed.

  If $job = 'V'$, then eigenvalues and eigenvectors are computed.

- $uplo$
  
  Must be 'U' or 'L'.

  If $uplo = 'U'$, $ap$ stores the packed upper triangular part of $A$.

  If $uplo = 'L'$, $ap$ stores the packed lower triangular part of $A$.

- $n$
  
  The order of the matrix $A$ ($n \geq 0$).

- $ap$
  
  Array $ap$ contains the packed upper or lower triangle of Hermitian matrix $A$, as specified by $uplo$.

  The size of $ap$ must be at least max$(1, n^{*}(n+1)/2)$.

- $ldz$
  
  The leading dimension of the output array $z$.

  Constraints:

  if $jobz = 'N'$, then $ldz \geq 1$;

  if $jobz = 'V'$, then $ldz \geq \text{max}(1, n)$.
Output Parameters

\( w \)

Array, size at least \( \max(1, n) \).
If \( \text{info} = 0 \), \( w \) contains the eigenvalues of the matrix \( A \) in ascending order.

\( z \)

Array \( z \) (size at least \( \max(1, \text{ldz} \times n) \)).
If \( \text{jobz} = 'V' \), then if \( \text{info} = 0 \), \( z \) contains the orthonormal eigenvectors of the matrix \( A \), with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i - 1] \).
If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

\( \text{ap} \)

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of \( A \).

Return Values

This function returns a value \( \text{info} \).
If \( \text{info} = 0 \), the execution is successful.
If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.
If \( \text{info} = i \), then the algorithm failed to converge; \( i \) indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

\( \text{?spevd} \)

Uses divide and conquer algorithm to compute all eigenvalues and (optionally) all eigenvectors of a real symmetric matrix held in packed storage.

Syntax

```c
lapack_int LAPACKE_sspevd (int matrix_layout, char jobz, char uplo, lapack_int n, float* ap, float* w, float* z, lapack_int ldz);
lapack_int LAPACKE_dspevd (int matrix_layout, char jobz, char uplo, lapack_int n, double* ap, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric matrix \( A \) (held in packed storage). In other words, it can compute the spectral factorization of \( A \) as:

\[ A = Z^*\Lambda Z^T. \]

Here \( \Lambda \) is a diagonal matrix whose diagonal elements are the eigenvalues \( \lambda_i \), and \( Z \) is the orthogonal matrix whose columns are the eigenvectors \( z_i \). Thus,

\[ A^*z_i = \lambda_i^*z_i \text{ for } i = 1, 2, \ldots, n. \]

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the \(QL \) or \(QR \) algorithm.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**jobz**
Must be 'N' or 'V'.
If jobz = 'N', then only eigenvalues are computed.
If jobz = 'V', then eigenvalues and eigenvectors are computed.

**uplo**
Must be 'U' or 'L'.
If uplo = 'U', ap stores the packed upper triangular part of A.
If uplo = 'L', ap stores the packed lower triangular part of A.

**n**
The order of the matrix A (n ≥ 0).

**ap**
ap contains the packed upper or lower triangle of symmetric matrix A, as specified by uplo.
The dimension of ap must be max(1, n*(n+1)/2)

**ldz**
The leading dimension of the output array z.
Constraints:
if jobz = 'N', then ldz ≥ 1;
if jobz = 'V', then ldz ≥ max(1, n).

Output Parameters

**w, z**
Arrays:
w, size at least max(1, n).
If info = 0, contains the eigenvalues of the matrix A in ascending order. See also info.
z (size max(1, ldz*n)).
If jobz = 'V', then this array is overwritten by the orthogonal matrix Z which contains the eigenvectors of A. If jobz = 'N', then z is not referenced.

**ap**
On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of A.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = i, then the algorithm failed to converge; i indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If info = -i, the i-th parameter had an illegal value.
Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $||E||_2 = O(\varepsilon) * ||A||_2$, where $\varepsilon$ is the machine precision.

The complex analogue of this routine is hpevd.

See also syevd for matrices held in full storage, and sbevd for banded matrices.

?hpevd

Uses divide and conquer algorithm to compute all eigenvalues and, optionally, all eigenvectors of a complex Hermitian matrix held in packed storage.

Syntax

```c
lapack_int LAPACKE_chpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_float* ap, float* w, lapack_complex_float* z, lapack_int ldz );
```

```c
lapack_int LAPACKE_zhpevd( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_complex_double* ap, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian matrix $A$ (held in packed storage). In other words, it can compute the spectral factorization of $A$ as: $A = Z \Lambda Z^H$.

Here $\Lambda$ is a real diagonal matrix whose diagonal elements are the eigenvalues $\lambda_i$, and $Z$ is the (complex) unitary matrix whose columns are the eigenvectors $z_i$. Thus,

$A z_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

Input Parameters

- `matrix_layout` specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `jobz` must be 'N' or 'V'.
  - If `jobz` = 'N', then only eigenvalues are computed.
  - If `jobz` = 'V', then eigenvalues and eigenvectors are computed.
- `uplo` must be 'U' or 'L'.
  - If `uplo` = 'U', `ap` stores the packed upper triangular part of $A$.
  - If `uplo` = 'L', `ap` stores the packed lower triangular part of $A$.
- `n` - the order of the matrix $A$ ($n \geq 0$).
- `ap` - contains the packed upper or lower triangle of Hermitian matrix $A$, as specified by `uplo`. 
The dimension of \( ap \) must be at least \( \max(1, n^* (n+1)/2) \).

\( ldz \)

The leading dimension of the output array \( z \).

Constraints:
- If \( jobz = 'N' \), then \( ldz \geq 1 \);
- If \( jobz = 'V' \), then \( ldz \geq \max(1, n) \).

Output Parameters

\( w \)

Array, size at least \( \max(1, n) \).
- If \( info = 0 \), contains the eigenvalues of the matrix \( A \) in ascending order.
- See also \( info \).

\( z \)

Array, size 1 if \( jobz = 'N' \) and \( \max(1, ldz*n) \) if \( jobz = 'V' \).
- If \( jobz = 'V' \), then this array is overwritten by the unitary matrix \( Z \) which contains the eigenvectors of \( A \).
- If \( jobz = 'N' \), then \( z \) is not referenced.

\( ap \)

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of \( A \).

Return Values

This function returns a value \( info \).
- If \( info=0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.
- If \( info = i \), then the algorithm failed to converge; \( i \) indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix \( A + E \) such that \( ||E||_2 = O(\epsilon) * ||A||_2 \), where \( \epsilon \) is the machine precision.

The real analogue of this routine is \( \text{spevd} \).

See also \( \text{heevd} \) for matrices held in full storage, and \( \text{hbevd} \) for banded matrices.

\( ? \text{spevx} \)

\text{Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix in packed storage.}

\textbf{Syntax}

```c
lapack_int LAPACKE_sspevx (int matrix_layout, char jobz, char range, char uplo,
  lapack_int n, float* ap, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
  lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);
```

```c
lapack_int LAPACKE_dspevx (int matrix_layout, char jobz, char range, char uplo,
  lapack_int n, double* ap, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
  lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```
Include Files
• mkl.h

Description
The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix \( A \) in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **jobz**: Must be 'N' or 'V'.
  - If \( job = 'N' \), then only eigenvalues are computed.
  - If \( job = 'V' \), then eigenvalues and eigenvectors are computed.
- **range**: Must be 'A' or 'V' or 'I'.
  - If \( range = 'A' \), the routine computes all eigenvalues.
  - If \( range = 'V' \), the routine computes eigenvalues \( w[i] \) in the half-open interval: \( vl < w[i] \leq vu \).
  - If \( range = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).
- **uplo**: Must be 'U' or 'L'.
  - If \( uplo = 'U' \), \( ap \) stores the packed upper triangular part of \( A \).
  - If \( uplo = 'L' \), \( ap \) stores the packed lower triangular part of \( A \).
- **n**: The order of the matrix \( A \) (\( n \geq 0 \)).
- **ap**: Array \( ap \) contains the packed upper or lower triangle of the symmetric matrix \( A \), as specified by \( uplo \).
  - The size of \( ap \) must be at least \( \max(1, n^*(n+1)/2) \).
- **vl, vu**: If \( range = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.
  - Constraint: \( vl < vu \).
  - If \( range = 'A' \) or 'I', \( vl \) and \( vu \) are not referenced.
- **il, iu**: If \( range = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.
  - Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il = 1 \) and \( iu = 0 \) if \( n = 0 \).
  - If \( range = 'A' \) or 'V', \( il \) and \( iu \) are not referenced.
- **abstol**: The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.
- **ldz**: The leading dimension of the output array \( z \).
Constraints:

if \( \text{jobz} = 'N' \), then \( ldz \geq 1 \);

if \( \text{jobz} = 'V' \), then \( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout.

Output Parameters

\( \text{ap} \)

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of \( A \).

The total number of eigenvalues found,

\( 0 \leq m \leq n \). If range = 'A', \( m = n \), if range = 'I', \( m = iu - il + 1 \), and if range = 'V' the exact value of \( m \) is not known in advance..

\( w, z \) Arrays:

\( w \), size at least \( \max(1, n) \).

If info = 0, contains the selected eigenvalues of the matrix \( A \) in ascending order.

\( z \)(size \( \max(1, ldz*m) \) for column major layout and \( \max(1, ldz*n) \) for row major layout).

If \( \text{jobz} = 'V' \), then if info = 0, the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i - 1] \).

If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

\( \text{ifail} \)

Array, size at least \( \max(1, n) \).

If \( \text{jobz} = 'V' \), then if info = 0, the first \( m \) elements of \( \text{ifail} \) are zero; if info > 0, the \( \text{ifail} \) contains the indices the eigenvectors that failed to converge.

If \( \text{jobz} = 'N' \), then \( \text{ifail} \) is not referenced.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the \( i \)-th parameter had an illegal value.

If info = i, then \( i \) eigenvectors failed to converge; their indices are stored in the array ifail.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to \( abstol + \varepsilon \max(|a|,|b|) \), where \( \varepsilon \) is the machine precision.
If $\text{abstol}$ is less than or equal to zero, then $\epsilon \|T\|_1$ will be used in its place, where $T$ is the tridiagonal matrix obtained by reducing $A$ to tridiagonal form. Eigenvalues will be computed most accurately when $\text{abstol}$ is set to twice the underflow threshold $2^*\text{?lamch}('S')$, not zero.

If this routine returns with $\text{info} > 0$, indicating that some eigenvectors did not converge, try setting $\text{abstol}$ to $2^*\text{?lamch}('S$).

$hpevx$

*Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix in packed storage.*

**Syntax**

```c
lapack_int LAPACKE_chpevx( int matrix_layout, char jobz, char range, char uplo,
    lapack_int n, lapack_complex_float* ap, float vl, float vu, lapack_int il, lapack_int iu,
    float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz,
    lapack_int* ifail );

lapack_int LAPACKE_zhpevx( int matrix_layout, char jobz, char range, char uplo,
    lapack_int n, lapack_complex_double* ap, double vl, double vu, lapack_int il, lapack_int iu,
    double abstol, lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz,
    lapack_int* ifail );
```

**Include Files**

- mkl.h

**Description**

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix $A$ in packed storage. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.

  - If $\text{job} = 'N'$, then only eigenvalues are computed.
  - If $\text{job} = 'V'$, then eigenvalues and eigenvectors are computed.

- **range**
  Must be 'A' or 'V' or 'I'.

  - If $\text{range} = 'A'$, the routine computes all eigenvalues.
  - If $\text{range} = 'V'$, the routine computes eigenvalues $w[i]$ in the half-open interval: $vl < w[i] \leq vu$.
  - If $\text{range} = 'I'$, the routine computes eigenvalues with indices $\text{il}$ to $\text{iu}$.

- **uplo**
  Must be 'U' or 'L'.

  - If $\text{uplo} = 'U'$, $ap$ stores the packed upper triangular part of $A$.
  - If $\text{uplo} = 'L'$, $ap$ stores the packed lower triangular part of $A$.

- **n**
  The order of the matrix $A (n \geq 0)$.
Array \textit{ap} contains the packed upper or lower triangle of the Hermitian matrix \textit{A}, as specified by \textit{uplo}.

The size of \textit{ap} must be at least \( \max(1, n^* (n+1)/2) \).

If \textit{range} = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl<vu \).

If \textit{range} = 'A' or 'I', \textit{vl} and \textit{vu} are not referenced.

If \textit{range} = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il=1 \) and \( iu=0 \) if \( n = 0 \).

If \textit{range} = 'A' or 'V', \textit{il} and \textit{iu} are not referenced.

The absolute error tolerance to which each eigenvalue is required. See \textit{Application notes} for details on error tolerance.

The leading dimension of the output array \textit{z}.

Constraints:

\begin{align*}
\text{if } \textit{jobz} = 'N', & \text{ then } \textit{ldz} \geq 1; \\
\text{if } \textit{jobz} = 'V', & \text{ then } \textit{ldz} \geq \max(1, n) \text{ for column major layout and } \textit{ldz} \geq \max(1, m) \text{ for row major layout.}
\end{align*}

Output Parameters

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form. The elements of the diagonal and the off-diagonal of the tridiagonal matrix overwrite the corresponding elements of \textit{A}.

The total number of eigenvalues found, \( 0 \leq m \leq n \).

Array, size at least \( \max(1, n) \).

If \textit{info} = 0, contains the selected eigenvalues of the matrix \textit{A} in ascending order.

Array \textit{z}(size \( \max(1, \textit{ldz} \times m) \) for column major layout and \( \max(1, \textit{ldz} \times n) \) for row major layout).

If \textit{jobz} = 'V', then if \textit{info} = 0, the first \( m \) columns of \textit{z} contain the orthonormal eigenvectors of the matrix \textit{A} corresponding to the selected eigenvalues, with the \( i \)-th column of \textit{z} holding the eigenvector associated with \textit{w}(\textit{i}).

If an eigenvector fails to converge, then that column of \textit{z} contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \textit{ifail}.

If \textit{jobz} = 'N', then \textit{z} is not referenced.
ifail

Array, size at least max(1, n).

If jobz = 'V', then if info = 0, the first \( m \) elements of ifail are zero; if info > 0, the ifail contains the indices the eigenvectors that failed to converge.

If jobz = 'N', then ifail is not referenced.

Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the \( i \)-th parameter had an illegal value.

If info = i, then \( i \) eigenvectors failed to converge; their indices are stored in the array ifail.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a,b]\) of width less than or equal to \( abstol + \epsilon \max(|a|,|b|) \), where \( \epsilon \) is the machine precision.

If abstol is less than or equal to zero, then \( \epsilon \times |T| \) will be used in its place, where \( T \) is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when abstol is set to twice the underflow threshold \( 2 \times \text{lamch}'(S)' \), not zero.

If this routine returns with info > 0, indicating that some eigenvectors did not converge, try setting abstol to \( 2 \times \text{lamch}'(S)' \).

?sbev

Computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.

Syntax

```c
lapack_int LAPACKE_ssbev (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, float* ab, lapack_int ldab, float* w, float* z, lapack_int ldz);
lapack_int LAPACKE_dsbev (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, double* ab, lapack_int ldab, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric band matrix \( A \).

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz

Must be 'N' or 'V'.

If jobz = 'N', then only eigenvalues are computed.

If jobz = 'V', then eigenvalues and eigenvectors are computed.
**uplo**
Must be 'U' or 'L'.
If uplo = 'U', ab stores the upper triangular part of A.
If uplo = 'L', ab stores the lower triangular part of A.

**n**
The order of the matrix A (n ≥ 0).

**kd**
The number of super- or sub-diagonals in A (kd ≥ 0).

**ab**
(a size at least max(1, ldab * n) for column major layout and at least max(1, ldab * (kd + 1)) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix A (as specified by uplo) in band storage format.

**ldab**
The leading dimension of ab; must be at least kd +1 for column major layout and n for row major layout.

**ldz**
The leading dimension of the output array z.
Constraints:
if jobz = 'N', then ldz ≥ 1;
if jobz = 'V', then ldz ≥ max(1, n).

**Output Parameters**

**w, z**
Arrays:
w, size at least max(1, n).
If info = 0, contains the eigenvalues of the matrix A in ascending order.
z(size max(1, ldz * n)).
If jobz = 'V', then if info = 0, z contains the orthonormal eigenvectors of the matrix A, with the i-th column of z holding the eigenvector associated with w[i - 1].
If jobz = 'N', then z is not referenced.

**ab**
On exit, this array is overwritten by the values generated during the reduction to tridiagonal form (see the description of ?sbtrd).

**Return Values**
This function returns a value info.
If info = 0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

**?hbev**
Computes all eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.
Syntax

```c
lapack_int LAPACKE_chbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w,
lapack_complex_float* z, lapack_int ldz );
```

```c
lapack_int LAPACKE_zhbev( int matrix_layout, char jobz, char uplo, lapack_int n,
lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w,
lapack_complex_double* z, lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix \( A \).

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `jobz` Must be 'N' or 'V'.
  - If `jobz` = 'N', then only eigenvalues are computed.
  - If `jobz` = 'V', then eigenvalues and eigenvectors are computed.
- `uplo` Must be 'U' or 'L'.
  - If `uplo` = 'U', `ab` stores the upper triangular part of \( A \).
  - If `uplo` = 'L', `ab` stores the lower triangular part of \( A \).
- `n` The order of the matrix \( A \) (\( n \geq 0 \)).
- `kd` The number of super- or sub-diagonals in \( A \) (\( kd \geq 0 \)).
- `ab` (size at least max(1, \( ldab * n \)) for column major layout and at least max(1, \( ldab * (kd + 1) \)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix \( A \) (as specified by `uplo`) in band storage format.
- `ldab` The leading dimension of `ab`; must be at least \( kd + 1 \) for column major layout and \( n \) for row major layout.
- `ldz` The leading dimension of the output array `z`.

Constraints:

- If `jobz` = 'N', then `ldz` ≥ 1;
- If `jobz` = 'V', then `ldz` ≥ max(1, \( n \)).

Output Parameters

- `w` Array, size at least max(1, \( n \)).
  - If `info` = 0, contains the eigenvalues in ascending order.
Array $z$ with size $\max(1, ldz*n)$.

If $jobz = 'V'$, then if $info = 0$, $z$ contains the orthonormal eigenvectors of the matrix $A$, with the $i$-th column of $z$ holding the eigenvector associated with $\lambda_{i-1}$.

If $jobz = 'N'$, then $z$ is not referenced.

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form (see the description of $\text{hbtrd}$).

Return Values
This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, then the algorithm failed to converge; $i$ indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.

$sbevd$

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric band matrix using divide and conquer algorithm.

Syntax

```c
lapack_int LAPACKE_ssbenvd (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, float* ab, lapack_int ldab, float* w, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dsbevd (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, double* ab, lapack_int ldab, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric band matrix $A$. In other words, it can compute the spectral factorization of $A$ as:

$$A = Z^T \Lambda Z$$

Here $\Lambda$ is a diagonal matrix whose diagonal elements are the eigenvalues $\lambda_i$, and $Z$ is the orthogonal matrix whose columns are the eigenvectors $z_i$. Thus,

$$A^T z_i = \lambda_i z_i$$

for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the $QL$ or $QR$ algorithm.

Input Parameters

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  
  Must be 'N' or 'V'.

If `jobz = 'N'`, then only eigenvalues are computed.
If `jobz = 'V'`, then eigenvalues and eigenvectors are computed.

`uplo` must be 'U' or 'L'.
If `uplo = 'U'`, `ab` stores the upper triangular part of `A`.
If `uplo = 'L'`, `ab` stores the lower triangular part of `A`.

`n` is the order of the matrix `A` (`n` ≥ 0).

`kd` is the number of super- or sub-diagonals in `A` (`kd` ≥ 0).

`ab` is an array containing either upper or lower triangular part of the symmetric matrix `A` (as specified by `uplo`) in band storage format.

`ldab` is the leading dimension of `ab`; must be at least `kd`+1 for column major layout and `n` for row major layout.

`ldz` is the leading dimension of the output array `z`.

Constraints:
- if `jobz = 'N'`, then `ldz` ≥ 1;
- if `jobz = 'V'`, then `ldz` ≥ max(1, `n`).

**Output Parameters**

`w`, `z` are arrays:

- `w`, size at least max(1, `n`).
  If `info = 0`, contains the eigenvalues of the matrix `A` in ascending order.
  See also `info`.
- `z` (size max(1, `ldz`*`n` if `job = 'V'` and at least 1 if `job = 'N'`)).
  If `job = 'V'`, then this array is overwritten by the orthogonal matrix `Z` which contains the eigenvectors of `A`. The `i`-th column of `Z` contains the eigenvector which corresponds to the eigenvalue `w[i - 1].`
  If `job = 'N'`, then `z` is not referenced.

`ab` is overwritten by the values generated during the reduction to tridiagonal form.

**Return Values**

This function returns a value `info`.

- If `info=0`, the execution is successful.
- If `info = i`, then the algorithm failed to converge; `i` indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
- If `info = -i`, the `i`-th parameter had an illegal value.
Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $A+E$ such that $||E||_2 = O(\epsilon) * ||A||_2$, where $\epsilon$ is the machine precision.

The complex analogue of this routine is hbevd.

See also syevd for matrices held in full storage, and spevd for matrices held in packed storage.

*?hbevd*

*Computes all eigenvalues and, optionally, all eigenvectors of a complex Hermitian band matrix using divide and conquer algorithm.*

**Syntax**

```c
lapack_int LAPACKE_chbevd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, lapack_complex_float* ab, lapack_int ldab, float* w, lapack_complex_float* z, lapack_int ldz);

lapack_int LAPACKE_zhbevd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int kd, lapack_complex_double* ab, lapack_int ldab, double* w, lapack_complex_double* z, lapack_int ldz);
```

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a complex Hermitian band matrix $A$. In other words, it can compute the spectral factorization of $A$ as: $A = Z^{*} \Lambda Z^{H}$.

Here $\Lambda$ is a real diagonal matrix whose diagonal elements are the eigenvalues $\lambda_i$, and $Z$ is the (complex) unitary matrix whose columns are the eigenvectors $z_i$. Thus, $A z_i = \lambda_i z_i$ for $i = 1, 2, \ldots, n$.

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

**Input Parameters**

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `jobz`
  Must be 'N' or 'V'.
  
  If `jobz` = 'N', then only eigenvalues are computed.
  
  If `jobz` = 'V', then eigenvalues and eigenvectors are computed.

- `uplo`
  Must be 'U' or 'L'.
  
  If `uplo` = 'U', $ab$ stores the upper triangular part of $A$.
  
  If `uplo` = 'L', $ab$ stores the lower triangular part of $A$.

- `n`
  The order of the matrix $A$ ($n \geq 0$).

- `kd`
  The number of super- or sub-diagonals in $A$.
(kd≥ 0).

**ab**

ab (size at least max(1, ldab*n) for column major layout and at least max(1, ldab*(kd + 1)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix A (as specified by uplo) in band storage format.

**ldab**

The leading dimension of ab; must be at least kd+1 for column major layout and n for row major layout.

**ldz**

The leading dimension of the output array z.

Constraints:

if jobz = 'N', then ldz≥ 1;
if jobz = 'V', then ldz≥ max(1, n).

**Output Parameters**

**w**

Array, size at least max(1, n).
If info = 0, contains the eigenvalues of the matrix A in ascending order. See also info.

**z**

Array, size max(1, ldz*n) if job = 'V' and at least 1 if job = 'N'.
If jobz = 'V', then this array is overwritten by the unitary matrix Z which contains the eigenvectors of A. The i-th column of Z contains the eigenvector which corresponds to the eigenvalue w[i - 1].
If jobz = 'N', then z is not referenced.

**ab**

On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

**Return Values**

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

**Application Notes**

The computed eigenvalues and eigenvectors are exact for a matrix $A + E$ such that $||E||_2 = O(\varepsilon) ||A||_2$, where $\varepsilon$ is the machine precision.

The real analogue of this routine is sbevd.
See also heevd for matrices held in full storage, and hpevd for matrices held in packed storage.

?sbevx

*Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix.*
Syntax
lapack_int LAPACKE_ssbevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, float* ab, lapack_int ldab, float* q, lapack_int ldq, float
vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
float* z, lapack_int ldz, lapack_int* ifail);

apack_int LAPACKE_dsbevx (int matrix_layout, char jobz, char range, char uplo,
lapack_int n, lapack_int kd, double* ab, lapack_int ldab, double* q, lapack_int ldq,
double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m,
double* w, double* z, lapack_int ldz, lapack_int* ifail);

Include Files
• mkl.h

Description
The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric band matrix $A$. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.
  If jobz = 'N', then only eigenvalues are computed.
  If jobz = 'V', then eigenvalues and eigenvectors are computed.

- **range**
  Must be 'A' or 'V' or 'I'.
  If range = 'A', the routine computes all eigenvalues.
  If range = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval: vl < w[i] ≤ vu.
  If range = 'I', the routine computes eigenvalues with indices in range il to iu.

- **uplo**
  Must be 'U' or 'L'.
  If uplo = 'U', ab stores the upper triangular part of $A$.
  If uplo = 'L', ab stores the lower triangular part of $A$.

- **n**
  The order of the matrix $A$ ($n ≥ 0$).

- **kd**
  The number of super- or sub-diagonals in $A$ ($kd ≥ 0$).

- **ab**
  Arrays:
  Array ab (size at least max(1, ldab*n) for column major layout and at least max(1, ldab*(kd + 1)) for row major layout) contains either upper or lower triangular part of the symmetric matrix $A$ (as specified by uplo) in band storage format.
ldab

The leading dimension of \( ab \); must be at least \( kd +1 \) for column major layout and \( n \) for row major layout.

vl, vu

If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If range = 'A' or 'I', vl and vu are not referenced.

il, iu

If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il = 1 \) and \( iu = 0 \) if \( n = 0 \).

If range = 'A' or 'V', il and iu are not referenced.

abstol

The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

ldq, ldz

The leading dimensions of the output arrays \( q \) and \( z \), respectively.

Constraints:

\( ldq \geq 1, \ ldz \geq 1 \);

If jobz = 'V', then \( ldq \geq \max(1, n) \) and \( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout.

Output Parameters

q

Array, size \( \max(1, \ ldz * n) \).

If jobz = 'V', the \( n \)-by-\( n \) orthogonal matrix is used in the reduction to tridiagonal form.

If jobz = 'N', the array \( q \) is not referenced.

m

The total number of eigenvalues found, \( 0 \leq m \leq n \).

If range = 'A', \( m = n \), if range = 'I', \( m = iu-il+1 \), and if range = 'V', the exact value of \( m \) is not known in advance.

w, z

Arrays:

\( w \), size at least \( \max(1, n) \). The first \( m \) elements of \( w \) contain the selected eigenvalues of the matrix \( A \) in ascending order.

\( z \), size at least \( \max(1, \ ldz * m) \) for column major layout and \( \max(1, \ ldz * n) \) for row major layout.

If jobz = 'V', then if info = 0, the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \).

If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

If jobz = 'N', then \( z \) is not referenced.
On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.

Array, size at least max(1, n).

If jobz = 'V', then if info = 0, the first m elements of ifail are zero; if info > 0, the ifail contains the indices the eigenvectors that failed to converge.

If jobz = 'N', then ifail is not referenced.

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info = i, then i eigenvectors failed to converge; their indices are stored in the array ifail.

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to abstol+ε*max(|a|,|b|), where ε is the machine precision.

If abstol is less than or equal to zero, then ε*||T||1 is used as tolerance, where T is the tridiagonal matrix obtained by reducing A to tridiagonal form. Eigenvalues will be computed most accurately when abstol is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with info > 0, indicating that some eigenvectors did not converge, try setting abstol to 2*?lamch('S').

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian band matrix.

The routine computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian band matrix A. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.
**Input Parameters**

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**jobz**
Must be 'N' or 'V'.
If job = 'N', then only eigenvalues are computed.
If job = 'V', then eigenvalues and eigenvectors are computed.

**range**
Must be 'A' or 'V' or 'I'.
If range = 'A', the routine computes all eigenvalues.
If range = 'V', the routine computes eigenvalues \( w[i] \) in the half-open interval: \( vl < w[i] \leq vu \).
If range = 'I', the routine computes eigenvalues with indices \( il \) to \( iu \).

**uplo**
Must be 'U' or 'L'.
If uplo = 'U', \( ab \) stores the upper triangular part of \( A \).
If uplo = 'L', \( ab \) stores the lower triangular part of \( A \).

**n**
The order of the matrix \( A \) (\( n \geq 0 \)).

**kd**
The number of super- or sub-diagonals in \( A \) (\( kd \geq 0 \)).

**ab**
\( ab \) (size at least max(1, \( ldab \times n \)) for column major layout and at least max(1, \( ldab \times (kd + 1) \)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix \( A \) (as specified by \( uplo \)) in band storage format.

**ldab**
The leading dimension of \( ab \); must be at least \( kd + 1 \) for column major layout and \( n \) for row major layout.

**vl, vu**
If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: \( vl < vu \).
If range = 'A' or 'I', \( vl \) and \( vu \) are not referenced.

**il, iu**
If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il = 1 \) and \( iu = 0 \) if \( n = 0 \).
If range = 'A' or 'V', \( il \) and \( iu \) are not referenced.

**abstol**
The absolute error tolerance to which each eigenvalue is required. See Application notes for details on error tolerance.

**ldq, ldz**
The leading dimensions of the output arrays \( q \) and \( z \), respectively.
Constraints:
\( ldq \geq 1, \ ldz \geq 1; \)
If \( \text{jobz} = 'V' \), then \( ldq \geq \max(1, n) \) and \( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout.

**Output Parameters**

- **q**: Array, size \( \max(1, ldz*n) \).
  - If \( \text{jobz} = 'V' \), the \( n \)-by-\( n \) unitary matrix is used in the reduction to tridiagonal form.
  - If \( \text{jobz} = 'N' \), the array \( q \) is not referenced.

- **m**: The total number of eigenvalues found, \( 0 \leq m \leq n \).
  - If \( \text{range} = 'A' \), \( m = n \); if \( \text{range} = 'I' \), \( m = iu-ii+1 \); and if \( \text{range} = 'V' \), the exact value of \( m \) is not known in advance.

- **w**: Array, size at least \( \max(1, n) \). The first \( m \) elements contain the selected eigenvalues of the matrix \( A \) in ascending order.

- **z**: Array \( z(\text{size at least} \ max(1, ldz*m) \text{ for column major layout and} \ max(1, ldz*n) \text{ for row major layout}) \).
  - If \( \text{jobz} = 'V' \), then if \( info = 0 \), the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \).
  - If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \( \text{ifail} \).
  - If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

- **ab**: On exit, this array is overwritten by the values generated during the reduction to tridiagonal form.
  - If \( \text{uplo} = 'U' \), the first superdiagonal and the diagonal of the tridiagonal matrix \( T \) are returned in rows \( kd \) and \( kd+1 \) of \( ab \), and if \( \text{uplo} = 'L' \), the diagonal and first subdiagonal of \( T \) are returned in the first two rows of \( ab \).

- **ifail**: Array, size at least \( \max(1, n) \).
  - If \( \text{jobz} = 'V' \), then if \( info = 0 \), the first \( m \) elements of \( \text{ifail} \) are zero; if \( info > 0 \), the \( \text{ifail} \) contains the indices of the eigenvectors that failed to converge.
  - If \( \text{jobz} = 'N' \), then \( \text{ifail} \) is not referenced.

**Return Values**

This function returns a value \( info \).

- If \( info = 0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.
- If \( info = i \), then \( i \) eigenvectors failed to converge; their indices are stored in the array \( \text{ifail} \).
Application Notes
An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a,b]\) of width less than or equal to \(abstol + \varepsilon \max( |a|, |b| )\), where \(\varepsilon\) is the machine precision.

If \(abstol\) is less than or equal to zero, then \(\varepsilon \|T\|_1\) will be used in its place, where \(T\) is the tridiagonal matrix obtained by reducing \(A\) to tridiagonal form. Eigenvalues will be computed most accurately when \(abstol\) is set to twice the underflow threshold \(2\times\text{?lamch('S')}\), not zero.

If this routine returns with \(info > 0\), indicating that some eigenvectors did not converge, try setting \(abstol\) to \(2\times\text{?lamch('S')}\).

\texttt{?stev}
\textit{Computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix.}

**Syntax**

```c
lapack_int LAPACKE_sstev (int matrix_layout, char jobz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);

lapack_int LAPACKE_dstev (int matrix_layout, char jobz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
```

**Include Files**

- mkl.h

**Description**
The routine computes all eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix \(A\).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.
  - If \(jobz = 'N'\), then only eigenvalues are computed.
  - If \(jobz = 'V'\), then eigenvalues and eigenvectors are computed.

- **n**
  The order of the matrix \(A\) (\(n \geq 0\)).

- **d, e**
  Arrays:
  - Array \(d\) contains the \(n\) diagonal elements of the tridiagonal matrix \(A\).
  - The size of \(d\) must be at least \(\max(1, n)\).
  - Array \(e\) contains the \(n-1\) subdiagonal elements of the tridiagonal matrix \(A\).
  - The size of \(e\) must be at least \(\max(1, n)\). The \(n\)-th element of this array is used as workspace.

- **ldz**
  The leading dimension of the output array \(z\); \(ldz \geq 1\). If \(jobz = 'V'\) then \(ldz \geq \max(1, n)\).
Output Parameters

\( d \)

On exit, if \( info = 0 \), contains the eigenvalues of the matrix \( A \) in ascending order.

\( z \)

Array, size (size max(1, \( ldz*n \))).

If \( jobz = 'V' \), then if \( info = 0 \), \( z \) contains the orthonormal eigenvectors of the matrix \( A \), with the \( i \)-th column of \( z \) holding the eigenvector associated with the eigenvalue returned in \( d[i - 1] \).

If \( job = 'N' \), then \( z \) is not referenced.

\( e \)

On exit, this array is overwritten with intermediate results.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = i \), then the algorithm failed to converge; \( i \) elements of \( e \) did not converge to zero.

### ?stevd

Computes all eigenvalues and, optionally, all eigenvectors of a real symmetric tridiagonal matrix using divide and conquer algorithm.

#### Syntax

```c
lapack_int LAPACKE_sstevd (int matrix_layout, char jobz, lapack_int n, float* d, float* e, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dstevd (int matrix_layout, char jobz, lapack_int n, double* d, double* e, double* z, lapack_int ldz);
```

#### Include Files

- mkl.h

#### Description

The routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix \( T \). In other words, the routine can compute the spectral factorization of \( T \) as: \( T = Z*\Lambda*Z^T \).

Here \( \Lambda \) is a diagonal matrix whose diagonal elements are the eigenvalues \( \lambda_i \), and \( Z \) is the orthogonal matrix whose columns are the eigenvectors \( z_i \). Thus, \( T*z_i = \lambda_i*z_i \) for \( i = 1, 2, \ldots, n \).

If the eigenvectors are requested, then this routine uses a divide and conquer algorithm to compute eigenvalues and eigenvectors. However, if only eigenvalues are required, then it uses the Pal-Walker-Kahan variant of the QL or QR algorithm.

There is no complex analogue of this routine.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**jobz**
Must be 'N' or 'V'.
If jobz = 'N', then only eigenvalues are computed.
If jobz = 'V', then eigenvalues and eigenvectors are computed.

**n**
The order of the matrix $T$ ($n \geq 0$).

**d, e**
Arrays:
d contains the $n$ diagonal elements of the tridiagonal matrix $T$.
The dimension of $d$ must be at least max(1, $n$).
e contains the $n$-1 off-diagonal elements of $T$.
The dimension of $e$ must be at least max(1, $n$). The $n$-th element of this array is used as workspace.

**ldz**
The leading dimension of the output array $z$. Constraints:
$ldz \geq 1$ if job = 'N';
$ldz \geq \text{max}(1, n)$ if job = 'V'.

Output Parameters

**d**
On exit, if info = 0, contains the eigenvalues of the matrix $T$ in ascending order.
See also info.

**z**
Array, size max(1, ldz*$n$) if jobz = 'V' and 1 if jobz = 'N'.
If jobz = 'V', then this array is overwritten by the orthogonal matrix $Z$ which contains the eigenvectors of $T$.
If jobz = 'N', then $z$ is not referenced.

**e**
On exit, this array is overwritten with intermediate results.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = i, then the algorithm failed to converge; $i$ indicates the number of elements of an intermediate tridiagonal form which did not converge to zero.
If info = -i, the $i$-th parameter had an illegal value.

Application Notes

The computed eigenvalues and eigenvectors are exact for a matrix $T+E$ such that $||E||_2 = O(\varepsilon)*||T||_2$, where $\varepsilon$ is the machine precision.
If $\lambda_i$ is an exact eigenvalue, and $\mu_i$ is the corresponding computed value, then
$|\mu_i - \lambda_i| \leq c(n) \varepsilon^* ||T||_2$
where \( c(n) \) is a modestly increasing function of \( n \).

If \( z_i \) is the corresponding exact eigenvector, and \( w_i \) is the corresponding computed vector, then the angle \( \theta(z_i, w_i) \) between them is bounded as follows:

\[
\theta(z_i, w_i) \leq c(n)^* \varepsilon^* \|T\|_2 / \min_{i \neq j} |\lambda_i - \lambda_j|.
\]

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

\textbf{?stevx}

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

\textbf{Syntax}

```c

lapack_int LAPACKE_sstevx (int matrix_layout, char jobz, char range, lapack_int n,
float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_dstevx (int matrix_layout, char jobz, char range, lapack_int n,
double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

\textbf{Include Files}

- mkl.h

\textbf{Description}

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix \( A \). Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

\textbf{Input Parameters}

\textit{matrix_layout} Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\textit{jobz} Must be 'N' or 'V'.

If \textit{job} = 'N', then only eigenvalues are computed.

If \textit{job} = 'V', then eigenvalues and eigenvectors are computed.

\textit{range} Must be 'A' or 'V' or 'I'.

If \textit{range} = 'A', the routine computes all eigenvalues.

If \textit{range} = 'V', the routine computes eigenvalues \( w[i] \) in the half-open interval: \( vl < w[i] \leq vu \).

If \textit{range} = 'I', the routine computes eigenvalues with indices \( il \) to \( iu \).

\textit{n} The order of the matrix \( A \) (\( n \geq 0 \)).

\textit{d, e} Arrays:

\( d \) contains the \( n \) diagonal elements of the tridiagonal matrix \( A \).

The dimension of \( d \) must be at least \( \max(1, n) \).
\( e \) contains the \( n-1 \) subdiagonal elements of \( A \).

The dimension of \( e \) must be at least \( \max(1, n-1) \). The \( n \)-th element of this array is used as workspace.

\( vl, vu \)

If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If \( \text{range} = 'A' \) or \( 'I' \), \( vl \) and \( vu \) are not referenced.

\( il, iu \)

If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il = 1 \) and \( iu = 0 \) if \( n = 0 \).

If \( \text{range} = 'A' \) or \( 'V' \), \( il \) and \( iu \) are not referenced.

\( abstol \)

\( ldz \)

The leading dimensions of the output array \( z \); \( ldz \geq 1 \). If \( \text{jobz} = 'V' \), then \( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout.

**Output Parameters**

\( m \)

The total number of eigenvalues found,

\( 0 \leq m \leq n \).

If \( \text{range} = 'A' \), \( m = n \), if \( \text{range} = 'I' \), \( m = iu-il+1 \), and if \( \text{range} = 'V' \) the exact value of \( m \) is unknown.

\( w, z \)

Arrays:

- \( w \), size at least \( \max(1, n) \).
- The first \( m \) elements of \( w \) contain the selected eigenvalues of the matrix \( A \) in ascending order.
- \( z \) (size at least \( \max(1, ldz*m) \) for column major layout and \( \max(1, ldz*n) \) for row major layout).

If \( \text{jobz} = 'V' \), then if \( \text{info} = 0 \), the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( A \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \).

If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \( \text{ifail} \).

If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

\( d, e \)

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

\( \text{ifail} \)

Array, size at least \( \max(1, n) \).
If $jobz = 'V'$, then if $info = 0$, the first $m$ elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, then $i$ eigenvectors failed to converge; their indices are stored in the array $ifail$.

Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to $abstol + \varepsilon \max(|a|, |b|)$, where $\varepsilon$ is the machine precision.

If $abstol$ is less than or equal to zero, then $\varepsilon |A|_1$ is used instead. Eigenvalues are computed most accurately when $abstol$ is set to twice the underflow threshold $2^* \text{?lamch}(\text'S')$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, set $abstol$ to $2^* \text{?lamch}(\text'S')$.

\textit{?stevr}

Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix using the Relatively Robust Representations.

Syntax

```c
lapack_int LAPACKE_sstevr (int matrix_layout, char jobz, char range, lapack_int n, float* d, float* e, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* isuppz);

lapack_int LAPACKE_dstevr (int matrix_layout, char jobz, char range, lapack_int n, double* d, double* e, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* isuppz);
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix $T$. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Whenever possible, the routine calls stemr to compute the eigenspectrum using Relatively Robust Representations. steqr computes eigenvalues by the dqds algorithm, while orthogonal eigenvectors are computed from various "good" $L^*D^*L^T$ representations (also known as Relatively Robust Representations). Gram-Schmidt orthogonalization is avoided as far as possible. More specifically, the various steps of the algorithm are as follows. For the $i$-th unreduced block of $T$:

a. Compute $T - \sigma_i = L_i^*D_i^*L_i^T$, such that $L_i^*D_i^*L_i^T$ is a relatively robust representation.

b. Compute the eigenvalues, $\lambda_j$, of $L_i^*D_i^*L_i^T$ to high relative accuracy by the dqds algorithm.
c. If there is a cluster of close eigenvalues, "choose" \( \sigma_i \) close to the cluster, and go to Step (a).

d. Given the approximate eigenvalue \( \lambda_j \) of \( \tilde{L}_i \tilde{D}_i \tilde{L}_i^T \), compute the corresponding eigenvector by forming a rank-revealing twisted factorization.

The desired accuracy of the output can be specified by the input parameter \( \text{abstol} \). The routine \( \text{?stevr} \) calls \( \text{stemr} \) when the full spectrum is requested on machines which conform to the IEEE-754 floating point standard. \( \text{?stevr} \) calls \( \text{stebz} \) and \( \text{stein} \) on non-IEEE machines and when partial spectrum requests are made.

### Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'N' or 'V'.

  - If \( \text{jobz} = 'N' \), then only eigenvalues are computed.
  - If \( \text{jobz} = 'V' \), then eigenvalues and eigenvectors are computed.

- **range**
  Must be 'A' or 'V' or 'I'.

  - If \( \text{range} = 'A' \), the routine computes all eigenvalues.
  - If \( \text{range} = 'V' \), the routine computes eigenvalues \( w[i] \) in the half-open interval:
    \( vl < w[i] \leq vu \).
  - If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).

  For \( \text{range} = 'V' \) or 'I' and \( iu-il < n-1 \), \( \text{stebz/dstebz} \) and \( \text{sstein/dstein} \) are called.

- **n**
  The order of the matrix \( T (n \geq 0) \).

- **d, e**
  Arrays:

  - \( d \) contains the \( n \) diagonal elements of the tridiagonal matrix \( T \).
  - \( e \) contains the \( n-1 \) subdiagonal elements of \( A \).

  The dimension of \( d \) must be at least \( \max(1, n) \).

  The dimension of \( e \) must be at least \( \max(1, n-1) \). The \( n \)-th element of this array is used as workspace.

- **vl, vu**
  If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.

  Constraint: \( vl < vu \).

  If \( \text{range} = 'A' \) or 'I', \( vl \) and \( vu \) are not referenced.

- **il, iu**
  If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.

  Constraint: \( 1 \leq il\leq iu \leq n \), if \( n > 0 \); \( il=1 \) and \( iu=0 \) if \( n = 0 \).

  If \( \text{range} = 'A' \) or 'V', \( il \) and \( iu \) are not referenced.

- **abstol**
  The absolute error tolerance to which each eigenvalue/eigenvector is required.
If \( \text{jobz} = \text{'V'} \), the eigenvalues and eigenvectors output have residual norms bounded by \( \text{abstol} \), and the dot products between different eigenvectors are bounded by \( \text{abstol} \). If \( \text{abstol} < n \cdot \text{eps} \cdot \|T\| \), then \( n \cdot \text{eps} \cdot \|T\| \) will be used in its place, where \( \text{eps} \) is the machine precision, and \( \|T\| \) is the 1-norm of the matrix \( T \). The eigenvalues are computed to an accuracy of \( \text{eps} \cdot \|T\| \) irrespective of \( \text{abstol} \). If high relative accuracy is important, set \( \text{abstol} \) to \( ?\text{lamch}\text{('S')} \).

\( \text{ldz} \)

The leading dimension of the output array \( z \).

Constraints:

- \( \text{ldz} \geq 1 \) if \( \text{jobz} = \text{'N'} \);
- \( \text{ldz} \geq \text{max}(1, n) \) for column major layout and \( \text{ldz} \geq \text{max}(1, m) \) for row major layout if \( \text{jobz} = \text{'V'} \).

\textbf{Output Parameters}

\( m \)

The total number of eigenvalues found,

\( 0 \leq m \leq n \). If \( \text{range} = \text{'A'} \), \( m = n \); if \( \text{range} = \text{'I'} \), \( m = iu-il+1 \), and if \( \text{range} = \text{'V'} \) the exact value of \( m \) is unknown.

\( w, z \)

Arrays:

- \( w \), size at least \( \text{max}(1, n) \).
- The first \( m \) elements of \( w \) contain the selected eigenvalues of the matrix \( T \) in ascending order.
- \( z \), size at least \( \text{max}(1, \text{ldz}\cdot m) \) for column major layout and \( \text{max}(1, \text{ldz}\cdot n) \) for row major layout.

If \( \text{jobz} = \text{'V'} \), then if \( \text{info} = 0 \), the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix \( T \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \).

If \( \text{jobz} = \text{'N'} \), then \( z \) is not referenced.

\( d, e \)

On exit, these arrays may be multiplied by a constant factor chosen to avoid overflow or underflow in computing the eigenvalues.

\( \text{isuppz} \)

Array, size at least \( 2 \cdot \text{max}(1, m) \).

The support of the eigenvectors in \( z \), i.e., the indices indicating the nonzero elements in \( z \). The \( i \)-th eigenvector is nonzero only in elements \( \text{isuppz}[2i-2] \) through \( \text{isuppz}[2i-1] \).

Implemented only for \( \text{range} = \text{'A'} \) or \( \text{'I'} \) and \( iu-il = n-1 \).

\textbf{Return Values}

This function returns a value \( \text{info} \). If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), an internal error has occurred.
Application Notes

Normal execution of the routine ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not handle NaNs and infinities in the IEEE standard default manner.

Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines

This section describes LAPACK driver routines used for solving nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Nonsymmetric Eigenproblems" lists all such driver routines.

Driver Routines for Solving Nonsymmetric Eigenproblems

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gees</td>
<td>Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.</td>
</tr>
<tr>
<td>geesx</td>
<td>Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.</td>
</tr>
<tr>
<td>geev</td>
<td>Computes the eigenvalues and left and right eigenvectors of a general matrix.</td>
</tr>
<tr>
<td>geevx</td>
<td>Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.</td>
</tr>
</tbody>
</table>

?gees

*Computes the eigenvalues and Schur factorization of a general matrix, and orders the factorization so that selected eigenvalues are at the top left of the Schur form.*

Syntax

```c
lapack_int LAPACKE_sgees( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2 select, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float* wr, float* wi, float* vs, lapack_int ldvs );

lapack_int LAPACKE_dgees( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2 select, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double* wr, double* wi, double* vs, lapack_int ldvs );

lapack_int LAPACKE_cgees( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1 select, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* sdim, lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs );

lapack_int LAPACKE_zgees( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1 select, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* sdim, lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs );
```

Include Files

- mkl.h

Description

The routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues, the real Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = Z^* T Z^H$. 
Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left. The leading columns of $Z$ then form an orthonormal basis for the invariant subspace corresponding to the selected eigenvalues.

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$
\begin{pmatrix}
    a & b \\
    c & a
\end{pmatrix}
$$

where $b^*c < 0$. The eigenvalues of such a block are $a \pm i\sqrt{bc}$.

A complex matrix is in Schur form if it is upper triangular.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobvs**
  Must be 'N' or 'V'.
  - If jobvs = 'N', then Schur vectors are not computed.
  - If jobvs = 'V', then Schur vectors are computed.

- **sort**
  Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.
  - If sort = 'N', then eigenvalues are not ordered.
  - If sort = 'S', eigenvalues are ordered (see select).

- **select**
  If sort = 'S', select is used to select eigenvalues to sort to the top left of the Schur form.
  - If sort = 'N', select is not referenced.

For real flavors:
An eigenvalue $wr[j]+\sqrt{-1}*wi[j]$ is selected if select($wr[j], wi[j]$) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

For complex flavors:
An eigenvalue $w[j]$ is selected if select($w[j]$) is true.

Note that a selected complex eigenvalue may no longer satisfy select($wr[j], wi[j]$) = 1 after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case info may be set to $n+2$ (see info below).

- **n**
  The order of the matrix $A$ ($n \geq 0$).

- **a**
  Arrays:
  - $a$ (size at least max(1, lda*$_n$)) is an array containing the $n$-by-$n$ matrix $A$.

- **lda**
  The leading dimension of the array $a$. Must be at least max(1, $n$).

- **ldvs**
  The leading dimension of the output array vs. Constraints:
ldvs ≥ 1;
ldvs ≥ max(1, n) if jobvs = 'V'.

Output Parameters

a
On exit, this array is overwritten by the real-Schur/Schur form T.

sdim
If sort = 'N', sdim = 0.
If sort = 'S', sdim is equal to the number of eigenvalues (after sorting) for which select is true.
Note that for real flavors complex conjugate pairs for which select is true for either eigenvalue count as 2.

wr, wi
Arrays, size at least max (1, n) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form T. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

w
Array, size at least max(1, n). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form T.

vs
Array vs (size at least max(1, ldvs*n)).
If jobvs = 'V', vs contains the orthogonal/unitary matrix Z of Schur vectors.
If jobvs = 'N', vs is not referenced.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
If info = i, and
i ≤ n:
the QR algorithm failed to compute all the eigenvalues; elements 1:ilo-1 and i+1:n of wr and wi (for real flavors) or w (for complex flavors) contain those eigenvalues which have converged; if jobvs = 'V', vs contains the matrix which reduces A to its partially converged Schur form;
i = n+1:
the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);
i = n+2:
after reordering, round-off changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy select = 1. This could also be caused by underflow due to scaling.
?geesx

Computes the eigenvalues and Schur factorization of a general matrix, orders the factorization and computes reciprocal condition numbers.

Syntax

```c
lapack_int LAPACKE_sgeesx( int matrix_layout, char jobvs, char sort, LAPACK_S_SELECT2 select, char sense, lapack_int n, float* a, lapack_int lda, lapack_int* sdim, float* wr, float* wi, float* vs, lapack_int ldvs, float* rconde, float* rcondv );
```

```c
lapack_int LAPACKE_dgeesx( int matrix_layout, char jobvs, char sort, LAPACK_D_SELECT2 select, char sense, lapack_int n, double* a, lapack_int lda, lapack_int* sdim, double* wr, double* wi, double* vs, lapack_int ldvs, double* rconde, double* rcondv );
```

```c
lapack_int LAPACKE_cgeesx( int matrix_layout, char jobvs, char sort, LAPACK_C_SELECT1 select, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int* sdim, lapack_complex_float* w, lapack_complex_float* vs, lapack_int ldvs, float* rconde, float* rcondv );
```

```c
lapack_int LAPACKE_zgeesx( int matrix_layout, char jobvs, char sort, LAPACK_Z_SELECT1 select, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int* sdim, lapack_complex_double* w, lapack_complex_double* vs, lapack_int ldvs, double* rconde, double* rcondv );
```

Include Files

- mkl.h

Description

The routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues, the real-Schur/Schur form T, and, optionally, the matrix of Schur vectors Z. This gives the Schur factorization $A = Z \cdot T \cdot Z^H$.

Optionally, it also orders the eigenvalues on the diagonal of the real-Schur/Schur form so that selected eigenvalues are at the top left; computes a reciprocal condition number for the average of the selected eigenvalues (rconde); and computes a reciprocal condition number for the right invariant subspace corresponding to the selected eigenvalues (rcondv). The leading columns of Z form an orthonormal basis for this invariant subspace.

For further explanation of the reciprocal condition numbers rconde and rcondv, see [LUG], Section 4.10 (where these quantities are called s and sep respectively).

A real matrix is in real-Schur form if it is upper quasi-triangular with 1-by-1 and 2-by-2 blocks. 2-by-2 blocks will be standardized in the form

$$
\begin{pmatrix}
    a & b \\
    c & a
\end{pmatrix}
$$

where $b \cdot c < 0$. The eigenvalues of such a block are $a \pm i \sqrt{bc}$.

A complex matrix is in Schur form if it is upper triangular.

Input Parameters

matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
**jobvs**

Must be 'N' or 'V'.

If jobvs = 'N', then Schur vectors are not computed.

If jobvs = 'V', then Schur vectors are computed.

**sort**

Must be 'N' or 'S'. Specifies whether or not to order the eigenvalues on the diagonal of the Schur form.

If sort = 'N', then eigenvalues are not ordered.

If sort = 'S', eigenvalues are ordered (see select).

**select**

If sort = 'S', select is used to select eigenvalues to sort to the top left of the Schur form.

If sort = 'N', select is not referenced.

*For real flavors:*

An eigenvalue \( wr[j] + \sqrt{-1} * wi[j] \) is selected if \( \text{select}(wr[j], wi[j]) \) is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

*For complex flavors:*

An eigenvalue \( w[j] \) is selected if \( \text{select}(w[j]) \) is true.

Note that a selected complex eigenvalue may no longer satisfy \( \text{select}(wr[j], wi[j]) = 1 \) after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case info may be set to \( n + 2 \) (see info below).

**sense**

Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If sense = 'N', none are computed;

If sense = 'E', computed for average of selected eigenvalues only;

If sense = 'V', computed for selected right invariant subspace only;

If sense = 'B', computed for both.

If sense is 'E', 'V', or 'B', then sort must equal 'S'.

**n**

The order of the matrix \( A \) (\( n \geq 0 \)).

**a**

Arrays:

\( a \) (size at least \( \max(1, lda* n) \)) is an array containing the \( n \)-by-\( n \) matrix \( A \).

**lda**

The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

**ldvs**

The leading dimension of the output array \( vs \). Constraints:

\( ldvs \geq 1 \);

\( ldvs \geq \max(1, n) \) if jobvs = 'V'.

**Output Parameters**

**a**

On exit, this array is overwritten by the real-Schur/Schur form \( T \).

**sdim**

If sort = 'N', \( sdim = 0 \).
If \( \text{sort} = 'S' \), \( sdim \) is equal to the number of eigenvalues (after sorting) for which \( \text{select} \) is true.

Note that for real flavors complex conjugate pairs for which \( \text{select} \) is true for either eigenvalue count as 2.

\( wr, wi \) Arrays, size at least \( \max(1, n) \) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues, in the same order that they appear on the diagonal of the output real-Schur form \( T \). Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

\( w \) Array, size at least \( \max(1, n) \). Contains the computed eigenvalues. The eigenvalues are stored in the same order as they appear on the diagonal of the output Schur form \( T \).

\( vs \) Array \( vs \) (size at least \( \max(1, \text{ldvs} \times n) \))

If \( \text{jobvs} = 'V' \), \( vs \) contains the orthogonal/unitary matrix \( Z \) of Schur vectors.

If \( \text{jobvs} = 'N' \), \( vs \) is not referenced.

\( rconde, rcondv \)

If \( \text{sense} = 'E' \) or \( 'B' \), \( rconde \) contains the reciprocal condition number for the average of the selected eigenvalues.

If \( \text{sense} = 'N' \) or \( 'V' \), \( rconde \) is not referenced.

If \( \text{sense} = 'V' \) or \( 'B' \), \( rcondv \) contains the reciprocal condition number for the selected right invariant subspace.

If \( \text{sense} = 'N' \) or \( 'E' \), \( rcondv \) is not referenced.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), and

\( i \leq n \):

the QR algorithm failed to compute all the eigenvalues; elements \( 1:ilo-1 \) and \( i+1:n \) of \( wr \) and \( wi \) (for real flavors) or \( w \) (for complex flavors) contain those eigenvalues which have converged; if \( \text{jobvs} = 'V' \), \( vs \) contains the transformation which reduces \( A \) to its partially converged Schur form;

\( i = n+1 \):

the eigenvalues could not be reordered because some eigenvalues were too close to separate (the problem is very ill-conditioned);

\( i = n+2 \):

after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Schur form no longer satisfy \( \text{select} = 1 \). This could also be caused by underflow due to scaling.

\?geev

*Computes the eigenvalues and left and right eigenvectors of a general matrix.*
Syntax

lapack_int LAPACKE_sgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n, 
float* a, lapack_int lda, float* wr, float* wi, float* vl, lapack_int ldvl, float* vr, 
lapack_int ldvr );

lapack_int LAPACKE_dgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n, 
double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl, double* 
vr, lapack_int ldvr );

lapack_int LAPACKE_cgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n, 
lapack_complex_float* a, lapack_int lda, lapack_complex_float* w, lapack_complex_float* 
vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );

lapack_int LAPACKE_zgeev( int matrix_layout, char jobvl, char jobvr, lapack_int n, 
lapack_complex_double* a, lapack_int lda, lapack_complex_double* w, 
lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int 
ldvr );

Include Files

• mkl.h

Description

The routine computes for an n-by-n real/complex nonsymmetric matrix A, the eigenvalues and, optionally, 
the left and/or right eigenvectors. The right eigenvector v of A satisfies

A*v = \lambda*v

where \lambda is its eigenvalue.

The left eigenvector u of A satisfies

u^H*A = \lambda*u^H

where u^H denotes the conjugate transpose of u. The computed eigenvectors are normalized to have 
Euclidean norm equal to 1 and largest component real.

Input Parameters

matrix_layout 

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) 
or column major (LAPACK_COL_MAJOR).

jobvl 

Must be 'N' or 'V'.

If jobvl = 'N', then left eigenvectors of A are not computed.

If jobvl = 'V', then left eigenvectors of A are computed.

jobvr 

Must be 'N' or 'V'.

If jobvr = 'N', then right eigenvectors of A are not computed.

If jobvr = 'V', then right eigenvectors of A are computed.

n 

The order of the matrix A (n ≥ 0).

a (size at least max(1, lda*n)) is an array containing the n-by-n matrix A.

lda 

The leading dimension of the array a. Must be at least max(1, n).

ldvl, ldvr 

The leading dimensions of the output arrays vl and vr, respectively.
Constraints:

\[ ldvl \geq 1; \quad ldvr \geq 1. \]

If \( jobvl = 'V' \), \( ldvl \geq \max(1, n) \);

If \( jobvr = 'V' \), \( ldvr \geq \max(1, n) \).

**Output Parameters**

\( a \)

On exit, this array is overwritten.

\( wr, wi \)

Arrays, size at least \( \max(1, n) \) each.

Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

\( w \)

Array, size at least \( \max(1, n) \).

Contains the computed eigenvalues.

\( vl, vr \)

Arrays:

- \( vl \) (size at least \( \max(1, ldvl*n) \)).
  
  If \( jobvl = 'N' \), \( vl \) is not referenced.

  **For real flavors:**
  
  If the \( j \)-th eigenvalue is real, the \( i \)-th component of the \( j \)-th eigenvector \( u_j \) is stored in \( vl[(i - 1) + (j - 1)*ldvl] \) for column major layout and in \( vl[(i - 1)*ldvl + (j - 1)] \) for row major layout.

  If the \( j \)-th and \((j+1)\)-st eigenvalues form a complex conjugate pair, then for \( i = \sqrt{-1} \), the \( k \)-th component of the \( j \)-th eigenvector \( u_j \) is stored in \( vl[(k - 1) + (j - 1)*ldvl] + i*vl[(k - 1) + (j + 1)*ldvl] \) for column major layout and as \( vl[(k - 1)*ldvl + (j - 1)] + i*vl[(k-1)*ldvl + j] \) for row major layout.

  Similarly, the \( k \)-th component of vector \( (j+1)u_{j+1} \) is stored in \( vl[(k - 1) + (j - 1)*ldvl] - i*vl[(k - 1) + (j + 1)*ldvl] \) for column major layout and as \( vl[(k - 1)*ldvl + (j - 1)] - i*vl[(k-1)*ldvl + j] \) for row major layout.

- \( vr \) (size at least \( \max(1, ldvr*n) \)).
  
  If \( jobvr = 'N' \), \( vr \) is not referenced.

  **For complex flavors:**
  
  The \( i \)-th component of the \( j \)-th eigenvector \( u_j \) is stored in \( vl[(i - 1) + (j - 1)*ldvl] \) for column major layout and in \( vl[(i - 1)*ldvl + (j - 1)] \) for row major layout.

  If the \( j \)-th eigenvalue is real, then the \( i \)-th component of the \( j \)-th eigenvector \( v_j \) is stored in \( vr[(i - 1) + (j - 1)*ldvr] \) for column major layout and in \( vr[(i - 1)*ldvr + (j - 1)] \) for row major layout.

  If the \( j \)-th and \((j+1)\)-st eigenvalues form a complex conjugate pair, then for \( i = \sqrt{-1} \), the \( k \)-th component of the \( j \)-th eigenvector \( v_j \) is stored in \( vr[(k - 1) + (j - 1)*ldvr] + i*vr[(k - 1) + (j + 1)*ldvr] \) for column major layout and as \( vr[(k - 1)*ldvr + (j - 1)] + i*vr[(k-1)*ldvr + j] \) for row major layout.
Similarly, the $k$-th component of vector $j + 1$ $v_{j+1}$ is $vr[(k - 1) + (j - 1)*ldvr] - i*vr[(k - 1) + j*ldvr]$ for column major layout and as $vr[(k - 1)*ldvr + (j - 1)] - i*vr[(k - 1)*ldvr + j]$ for row major layout.

For complex flavors:

The $i$-th component of the $j$-th eigenvector $v_j$ is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, the QR algorithm failed to compute all the eigenvalues, and no eigenvectors have been computed; elements $i+1:n$ of $wr$ and $wi$ (for real flavors) or $w$ (for complex flavors) contain those eigenvalues which have converged.

?geevx

Computes the eigenvalues and left and right eigenvectors of a general matrix, with preliminary matrix balancing, and computes reciprocal condition numbers for the eigenvalues and right eigenvectors.

Syntax

```c
lapack_int LAPACKE_sgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, float* a, lapack_int lda, float* wr, float* wi, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* scale, float* abnrm, float* rconde, float* rcondv );
```

```c
lapack_int LAPACKE_dgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, double* a, lapack_int lda, double* wr, double* wi, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* scale, double* abnrm, double* rconde, double* rcondv );
```

```c
lapack_int LAPACKE_cgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* w, lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* scale, float* abnrm, float* rconde, float* rcondv );
```

```c
lapack_int LAPACKE_zgeevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* w, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* scale, double* abnrm, double* rconde, double* rcondv );
```

Include Files

- mkl.h

Description
The routine computes for an \( n \times n \) real/complex nonsymmetric matrix \( A \), the eigenvalues and, optionally, the left and/or right eigenvectors.

Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (\( \text{ilo, ihi, scale, and abnrms} \)), reciprocal condition numbers for the eigenvalues (\( \text{rconde} \)), and reciprocal condition numbers for the right eigenvectors (\( \text{rconde}_v \)).

The right eigenvector \( v \) of \( A \) satisfies
\[
A v = \lambda v
\]
where \( \lambda \) is its eigenvalue.

The left eigenvector \( u \) of \( A \) satisfies
\[
u^H A = \lambda u^H
\]
where \( u^T \) denotes the conjugate transpose of \( u \). The computed eigenvectors are normalized to have Euclidean norm equal to 1 and largest component real.

Balancing a matrix means permuting the rows and columns to make it more nearly upper triangular, and applying a diagonal similarity transformation \( D^* A D^{-1} \), where \( D \) is a diagonal matrix, to make its rows and columns closer in norm and the condition numbers of its eigenvalues and eigenvectors smaller. The computed reciprocal condition numbers correspond to the balanced matrix. Permuting rows and columns will not change the condition numbers in exact arithmetic) but diagonal scaling will. For further explanation of balancing, see [LUG], Section 4.10.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **balanc**
  Must be 'N', 'P', 'S', or 'B'. Indicates how the input matrix should be diagonally scaled and/or permuted to improve the conditioning of its eigenvalues.
  - If \( \text{balanc} = 'N' \), do not diagonally scale or permute;
  - If \( \text{balanc} = 'P' \), perform permutations to make the matrix more nearly upper triangular. Do not diagonally scale;
  - If \( \text{balanc} = 'S' \), diagonally scale the matrix, i.e. replace \( A \) by \( D^* A D^{-1} \), where \( D \) is a diagonal matrix chosen to make the rows and columns of \( A \) more equal in norm. Do not permute;
  - If \( \text{balanc} = 'B' \), both diagonally scale and permute \( A \).

  Computed reciprocal condition numbers will be for the matrix after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

- **jobvl**
  Must be 'N' or 'V'.
  - If \( \text{jobvl} = 'N' \), left eigenvectors of \( A \) are not computed;
  - If \( \text{jobvl} = 'V' \), left eigenvectors of \( A \) are computed.
  - If \( \text{sense} = 'E' \) or 'B', then \( \text{jobvl} \) must be 'V'.

- **jobvr**
  Must be 'N' or 'V'.
  - If \( \text{jobvr} = 'N' \), right eigenvectors of \( A \) are not computed;
  - If \( \text{jobvr} = 'V' \), right eigenvectors of \( A \) are computed.
If \( \text{sense} = 'E' \) or \'B', then \( \text{jobvr} \) must be \'V'.

\( \text{sense} \)

Must be \'N', \'E', \'V', \) or \'B'. Determines which reciprocal condition number are computed.

If \( \text{sense} = 'N' \), none are computed;
If \( \text{sense} = 'E' \), computed for eigenvalues only;
If \( \text{sense} = 'V' \), computed for right eigenvectors only;
If \( \text{sense} = 'B' \), computed for eigenvalues and right eigenvectors.

If \( \text{sense} \) is \'E' or \'B', both left and right eigenvectors must also be computed (\( \text{jobvl} = 'V' \) and \( \text{jobvr} = 'V' \)).

\( n \)
The order of the matrix \( A (n\geq 0) \).

\( a \)
Arrays:
\( a \) (size at least \( \max(1, lda*n) \)) is an array containing the \( n \)-by-\( n \) matrix \( A \).

\( lda \)
The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

\( ldvl, ldvr \)
The leading dimensions of the output arrays \( vl \) and \( vr \), respectively.

Constraints:
\( ldvl \geq 1; ldvr \geq 1. \)

If \( \text{jobvl} = 'V' \), \( ldvl \geq \max(1, n) \);
If \( \text{jobvr} = 'V' \), \( ldvr \geq \max(1, n) \).

Output Parameters

\( a \)
On exit, this array is overwritten.
If \( \text{jobvl} = 'V' \) or \( \text{jobvr} = 'V' \), it contains the real-Schur/Schur form of the balanced version of the input matrix \( A \).

\( wr, wi \)
Arrays, size at least \( \max(1, n) \) each. Contain the real and imaginary parts, respectively, of the computed eigenvalues. Complex conjugate pairs of eigenvalues appear consecutively with the eigenvalue having positive imaginary part first.

\( w \)
Array, size at least \( \max(1, n) \). Contains the computed eigenvalues.

\( vl, vr \)
Arrays:
\( vl \) (size at least \( \max(1, ldvl*n) \)).
If \( \text{jobvl} = 'N' \), \( vl \) is not referenced.

For real flavors:
If the \( j \)-th eigenvalue is real, the \( i \)-th component of the \( j \)-th eigenvector \( u_j \) is stored in \( vl[(i-1)+(j-1)*ldvl] \) for column major layout and in \( vl[(i-1)*ldvl + (j-1)] \) for row major layout..

If the \( j \)-th and \( (j+1) \)-st eigenvalues form a complex conjugate pair, then for \( i = \sqrt{-1} \), the \( k \)-th component of the \( j \)-th eigenvector \( u_j \) is \( vl[(k-1)+(j-1)*ldvl] + i*vl[(k-1)+(j-1)*ldvl] \) for column major layout and as \( vl[(k-1)*ldvl + (j-1)] + i*vl[(k-1)*ldvl + j] \) for row major layout.
Similarly, the $k$-th component of vector $(j+1) u_{j+1}$ is $vl[(k - 1) + (j - 1)*ldvl] - i*vl[(k - 1) + j*ldvl]$ for column major layout and as $vl[(k - 1)*ldvl + (j - 1)] - i*vl[(k - 1)*ldvl + j]$ for row major layout.

For complex flavors:
The $i$-th component of the $j$-th eigenvector $u_j$ is stored in $vl[(i - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(i - 1)*ldvl + (j - 1)]$ for row major layout.

For real flavors:
If the $j$-th eigenvalue is real, then the $i$-th component of $j$-th eigenvector $v_j$ is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout.

For complex flavors:
The $i$-th component of the $j$-th eigenvector $v_j$ is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout.

For complex flavors:
The $i$-th component of the $j$-th eigenvector $v_j$ is stored in $vr[(i - 1) + (j - 1)*ldvr]$ for column major layout and in $vr[(i - 1)*ldvr + (j - 1)]$ for row major layout.

$i_l o, i_h i$

$i_l o$ and $i_h i$ are integer values determined when $A$ was balanced.

The balanced $A(i,j) = 0$ if $i > j$ and $j = 1, \ldots, i_l o - 1$ or $i = i_h i + 1, \ldots, n$.

If $b a l a n c = ' N ' o r ' S ' , i_l o = 1$ and $i_h i = n$.

$s c a l e$

Array, size at least max(1, $n$). Details of the permutations and scaling factors applied when balancing $A$.

If $P[j - 1]$ is the index of the row and column interchanged with row and column $J$, and $D[j - 1]$ is the scaling factor applied to row and column $J$, then

$s c a l e[j - 1] = P[j - 1], f o r j = 1, \ldots, i_l o - 1$

$= D[j - 1], f o r j = i_l o, \ldots, i_h i$

$= P[j - 1] f o r j = i_h i + 1, \ldots, n$.

The order in which the interchanges are made is $n$ to $i_h i + 1$, then 1 to $i_l o - 1$.

$a b n r m$

The one-norm of the balanced matrix (the maximum of the sum of absolute values of elements of any column).

$r c o n d _ e, r c o n d _ v$

Arrays, size at least max(1, $n$) each.

$r c o n d _ e[j - 1]$ is the reciprocal condition number of the $j$-th eigenvalue.
\[ r\text{condv}[j - 1] \] is the reciprocal condition number of the \( j \)-th right eigenvector.

**Return Values**

This function returns a value \( info \).

- If \( info=0 \), the execution is successful.
- If \( info = -i \), the \( i \)-th parameter had an illegal value.
- If \( info = i \), the QR algorithm failed to compute all the eigenvalues, and no eigenvectors or condition numbers have been computed; elements 1:ilo-1 and i+1:n of \( wr \) and \( wi \) (for real flavors) or \( w \) (for complex flavors) contain eigenvalues which have converged.

**Singular Value Decomposition: LAPACK Driver Routines**

Table "Driver Routines for Singular Value Decomposition" lists the LAPACK driver routines that perform singular value decomposition.

**Driver Routines for Singular Value Decomposition**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>?gesvd</td>
<td>Computes the singular value decomposition of a general rectangular matrix.</td>
</tr>
<tr>
<td>?gesdd</td>
<td>Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.</td>
</tr>
<tr>
<td>?gejsv</td>
<td>Computes the singular value decomposition of a real matrix using a preconditioned Jacobi SVD method.</td>
</tr>
<tr>
<td>?gesvj</td>
<td>Computes the singular value decomposition of a real matrix using Jacobi plane rotations.</td>
</tr>
<tr>
<td>?ggsvd</td>
<td>Computes the generalized singular value decomposition of a pair of general rectangular matrices.</td>
</tr>
<tr>
<td>?gesvdx</td>
<td>Computes the SVD and left and right singular vectors for a matrix.</td>
</tr>
<tr>
<td>?bdsvdx</td>
<td>Computes the SVD of a bidiagonal matrix.</td>
</tr>
</tbody>
</table>

**Singular Value Decomposition - LAPACK Computational Routines**

?gesvd

*Computes the singular value decomposition of a general rectangular matrix.*

**Syntax**

```c
lapack_int LAPACKE_sgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                         lapack_int n, float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt,
                         lapack_int ldvt, float* superb );

lapack_int LAPACKE_dgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                         lapack_int n, double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double* vt,
                         lapack_int ldvt, double* superb );

lapack_int LAPACKE_cgesvd( int matrix_layout, char jobu, char jobvt, lapack_int m,
                         lapack_int n, lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float* u,
                         lapack_int ldu, lapack_complex_float* vt, lapack_int ldvt, float* superb );
```
lapack_int LAPACKE_zgesvd ( int matrix_layout, char jobu, char jobvt, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, double* s, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt, double* superb );

Include Files

- mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real/complex m-by-n matrix A, optionally computing the left and/or right singular vectors. The SVD is written as

\[ A = U \Sigma V^T \] for real routines

\[ A = U \Sigma V^H \] for complex routines

where \( \Sigma \) is an \( m \times n \) matrix which is zero except for its \( \min(m,n) \) diagonal elements, \( U \) is an \( m \times m \) orthogonal/unitary matrix, and \( V \) is an \( n \times n \) orthogonal/unitary matrix. The diagonal elements of \( \Sigma \) are the singular values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m, n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

The routine returns \( V^T \) (for real flavors) or \( V^H \) (for complex flavors), not \( V \).

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobu**: Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix \( U \).
  - If \( jobu = 'A' \), all \( m \) columns of \( U \) are returned in the array \( u \);
  - if \( jobu = 'S' \), the first \( \min(m, n) \) columns of \( U \) (the left singular vectors) are returned in the array \( u \);
  - if \( jobu = 'O' \), the first \( \min(m, n) \) columns of \( U \) (the left singular vectors) are overwritten on the array \( a \);
  - if \( jobu = 'N' \), no columns of \( U \) (no left singular vectors) are computed.

- **jobvt**: Must be 'A', 'S', 'O', or 'N'. Specifies options for computing all or part of the matrix \( V^T/V^H \).
  - If \( jobvt = 'A' \), all \( n \) rows of \( V^T/V^H \) are returned in the array \( vt \);
  - if \( jobvt = 'S' \), the first \( \min(m, n) \) rows of \( V^T/V^H \) (the right singular vectors) are returned in the array \( vt \);
  - if \( jobvt = 'O' \), the first \( \min(m, n) \) rows of \( V^T/V^H \) (the right singular vectors) are overwritten on the array \( a \);
  - if \( jobvt = 'N' \), no rows of \( V^T/V^H \) (no right singular vectors) are computed.

- **m**: The number of rows of the matrix \( A \) (\( m \geq 0 \)).

- **n**: The number of columns in \( A \) (\( n \geq 0 \)).

Arrays:
A (size at least \(\max(1, lda*n)\) for column major layout and \(\max(1, lda*m)\) for row major layout) is an array containing the \(m\)-by-\(n\) matrix \(A\).

**lda**

The leading dimension of the array \(a\).

Must be at least \(\max(1, m)\) for column major layout and at least \(\max(1, n)\) for row major layout.

**ldu, ldvt**

The leading dimensions of the output arrays \(u\) and \(vt\), respectively.

**Constraints:**

\(ldu \geq 1; ldvt \geq 1\).

If \(jobu = 'A'\), \(ldu \geq m\); 

If \(jobu = 'S'\), \(ldu \geq m\) for column major layout and \(ldu \geq \min(m, n)\) for row major layout;

If \(jobvt = 'A'\), \(ldvt \geq n\); 

If \(jobvt = 'S'\), \(ldvt \geq \min(m, n)\) for column major layout and \(ldvt \geq n\) for row major layout.

**Output Parameters**

**\(a\)**

On exit,

If \(jobu = 'O'\), \(a\) is overwritten with the first \(\min(m, n)\) columns of \(U\) (the left singular vectors stored columnwise);

If \(jobvt = 'O'\), \(a\) is overwritten with the first \(\min(m, n)\) rows of \(V^T/V^H\) (the right singular vectors stored rowwise);

If \(jobu \neq 'O' \text{ and } jobvt \neq 'O'\), the contents of \(a\) are destroyed.

**\(s\)**

Array, size at least \(\max(1, \min(m, n))\). Contains the singular values of \(A\) sorted so that \(s[i] \geq s[i + 1]\).

**Arrays:**

Array \(u\) minimum size:

<table>
<thead>
<tr>
<th></th>
<th>Column major layout</th>
<th>Row major layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>(jobu = 'A')</td>
<td>(\max(1, ldu*m))</td>
<td>(\max(1, ldu*m))</td>
</tr>
<tr>
<td>(jobu = 'S')</td>
<td>(\max(1, ldu*\min(m, n)))</td>
<td>(\max(1, ldu*m))</td>
</tr>
</tbody>
</table>

If \(jobu = 'A'\), \(u\) contains the \(m\)-by-\(m\) orthogonal/unitary matrix \(U\).

If \(jobu = 'S'\), \(u\) contains the first \(\min(m, n)\) columns of \(U\) (the left singular vectors stored column-wise).

If \(jobu = 'N' \text{ or } 'O'\), \(u\) is not referenced.

Array \(v\) minimum size:

<table>
<thead>
<tr>
<th></th>
<th>Column major layout</th>
<th>Row major layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>(jobvt = 'A')</td>
<td>(\max(1, ldvt*n))</td>
<td>(\max(1, ldvt*n))</td>
</tr>
</tbody>
</table>
If \( \text{jobvt} = 'A' \), \( vt \) contains the \( n \)-by-\( n \) orthogonal/unitary matrix \( V^T/V^H \).

If \( \text{jobvt} = 'S' \), \( vt \) contains the first \( \min(m, n) \) rows of \( V^T/V^H \) (the right singular vectors stored row-wise).

If \( \text{jobvt} = 'N' \) or 'O', \( vt \) is not referenced.

If \( ?bdsqr \) does not converge (indicated by the return value \( \text{info} > 0 \)), on exit \( \text{superb}(0:\min(m,n)-2) \) contains the unconverged superdiagonal elements of an upper bidiagonal matrix \( B \) whose diagonal is in \( s \) (not necessarily sorted). \( B \) satisfies \( A = u*B*V^T \) (real flavors) or \( A = u*B*V^H \) (complex flavors), so it has the same singular values as \( A \), and singular vectors related by \( u \) and \( vt \).

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), then if \( ?bdsqr \) did not converge, \( i \) specifies how many superdiagonals of the intermediate bidiagonal form \( B \) did not converge to zero (see the description of the \( \text{superb} \) parameter for details).

**?gesdd**

*Computes the singular value decomposition of a general rectangular matrix using a divide and conquer method.*

**Syntax**

```c
lapack_int LAPACKE_sgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
float* a, lapack_int lda, float* s, float* u, lapack_int ldu, float* vt, lapack_int ldvt );

lapack_int LAPACKE_dgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
double* a, lapack_int lda, double* s, double* u, lapack_int ldu, double* vt, lapack_int ldvt );

lapack_int LAPACKE_cgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_float* a, lapack_int lda, float* s, lapack_complex_float* u, lapack_int ldu, lapack_complex_float* vt, lapack_int ldvt );

lapack_int LAPACKE_zgesdd( int matrix_layout, char jobz, lapack_int m, lapack_int n,
lapack_complex_double* a, lapack_int lda, double* s, lapack_complex_double* u, lapack_int ldu, lapack_complex_double* vt, lapack_int ldvt );
```

**Include Files**

- mkl.h

**Description**
The routine computes the singular value decomposition (SVD) of a real/complex \( m \)-by-\( n \) matrix \( A \), optionally computing the left and/or right singular vectors. If singular vectors are desired, it uses a divide-and-conquer algorithm. The SVD is written

\[
A = U^* \Sigma V^T \quad \text{for real routines,}
\]

\[
A = U^* \Sigma V^H \quad \text{for complex routines,}
\]

where \( \Sigma \) is an \( m \)-by-\( n \) matrix which is zero except for its \( \min(m,n) \) diagonal elements, \( U \) is an \( m \)-by-\( m \) orthogonal/unitary matrix, and \( V \) is an \( n \)-by-\( n \) orthogonal/unitary matrix. The diagonal elements of \( \Sigma \) are the singular values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m,n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

Note that the routine returns \( v^t = V^T \) (for real flavors) or \( v^t = V^H \) (for complex flavors), not \( V \).

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobz**
  Must be 'A', 'S', 'O', or 'N'.

  Specifies options for computing all or part of the matrices \( U \) and \( V \).

  If \( \text{jobz} = \text{'A'} \), all \( m \) columns of \( U \) and all \( n \) rows of \( V^T \) or \( V^H \) are returned in the arrays \( u \) and \( v^t \);

  if \( \text{jobz} = \text{'S'} \), the first \( \min(m,n) \) columns of \( U \) and the first \( \min(m,n) \) rows of \( V^T \) or \( V^H \) are returned in the arrays \( u \) and \( v^t \);

  if \( \text{jobz} = \text{'O'} \), then

  if \( m \geq n \), the first \( n \) columns of \( U \) are overwritten in the array \( a \) and all rows of \( V^T \) or \( V^H \) are returned in the array \( v^t \);

  if \( m < n \), all columns of \( U \) are returned in the array \( u \) and the first \( m \) rows of \( V^T \) or \( V^H \) are overwritten in the array \( a \);

  if \( \text{jobz} = \text{'N'} \), no columns of \( U \) or rows of \( V^T \) or \( V^H \) are computed.

- **m**
  The number of rows of the matrix \( A \) (\( m \geq 0 \)).

- **n**
  The number of columns in \( A \) (\( n \geq 0 \)).

- **a**
  \( \text{a(size max(1, lda*n))} \) for column major layout and \( \text{max(1, lda*m)} \) for row major layout) is an array containing the \( m \)-by-\( n \) matrix \( A \).

- **lda**
  The leading dimension of the array \( a \). Must be at least \( \text{max(1, m)} \) for column major layout and at least \( \text{max(1, n)} \) for row major layout.

- **ldu, ldvt**
  The leading dimensions of the output arrays \( u \) and \( v^t \), respectively.

  The minimum size of \( ldu \) is

<table>
<thead>
<tr>
<th>( \text{jobz} )</th>
<th>( m \geq n )</th>
<th>( m &lt; n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>'A'</td>
<td>( m )</td>
<td>( m )</td>
</tr>
<tr>
<td>'S'</td>
<td>( m ) for column major layout; ( n ) for row major layout</td>
<td>( m ) for column major layout; ( n ) for row major layout</td>
</tr>
</tbody>
</table>
Output Parameters

On exit:

If jobz = 'O', then if m ≥ n, a is overwritten with the first n columns of U (the left singular vectors, stored columnwise). If m < n, a is overwritten with the first m rows of Vᵀ (the right singular vectors, stored rowwise);

If jobz≠'O', the contents of a are destroyed.

Array, size at least max(1, min(m, n)). Contains the singular values of A sorted so that s(i) ≥ s(i+1).

Arrays:

Array u is of size:

<table>
<thead>
<tr>
<th>jobz</th>
<th>m≥n</th>
<th>m &lt; n</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>'A'</td>
<td>max(1, ldu*m)</td>
<td>max(1, ldu*m)</td>
</tr>
<tr>
<td>'S'</td>
<td>max(1, ldu<em>n) for column major layout; max(1, ldu</em>m) for row major layout</td>
<td>max(1, ldu*m)</td>
</tr>
<tr>
<td>'O'</td>
<td>1</td>
<td>max(1, ldu*m)</td>
</tr>
</tbody>
</table>

If jobz = 'A' or jobz = 'O' and m < n, u contains the m-by-m orthogonal/unitary matrix U.

If jobz = 'S', u contains the first min(m, n) columns of U (the left singular vectors, stored columnwise).

If jobz = 'O' and m≥n, or jobz = 'N', u is not referenced.

Array vt is of size:
<table>
<thead>
<tr>
<th>jobz</th>
<th>( m \geq n )</th>
<th>( m &lt; n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>'A'</td>
<td>( \max(1, \text{ldvt}*n) )</td>
<td>( \max(1, \text{ldvt}*n) )</td>
</tr>
<tr>
<td>'S'</td>
<td>( \max(1, \text{ldvt}*n) )</td>
<td>( \max(1, \text{ldvt}*n) ) for column major layout; ( \max(1, \text{ldvt}*m) ) for row major layout;</td>
</tr>
<tr>
<td>'O'</td>
<td>( \max(1, \text{ldvt}*n) )</td>
<td>1</td>
</tr>
</tbody>
</table>

If \( \text{jobz} = 'A' \) or \( \text{jobz} = 'O' \) and \( m \geq n \), \( \text{vt} \) contains the \( n \)-by-\( n \) orthogonal/unitary matrix \( V^T \).

If \( \text{jobz} = 'S' \), \( \text{vt} \) contains the first \( \min(m, n) \) rows of \( V^T \) (the right singular vectors, stored rowwise).

If \( \text{jobz} = 'O' \) and \( m < n \), or \( \text{jobz} = 'N' \), \( \text{vt} \) is not referenced.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), then \( \text{?bdsdc} \) did not converge, updating process failed.

**?gejsv**

*Computes the singular value decomposition using a preconditioned Jacobi SVD method.*

**Syntax**

```c
lapack_int LAPACKE_sgejsv (int matrix_layout, char joba, char jobu, char jobv, char jobr, char jobt, char jobp, lapack_int m, lapack_int n, float * a, lapack_int lda, float * sva, float * u, lapack_int ldu, float * v, lapack_int ldv, float * stat, lapack_int * istat);

lapack_int LAPACKE_dgejsv (int matrix_layout, char joba, char jobu, char jobv, char jobr, char jobt, char jobp, lapack_int m, lapack_int n, double * a, lapack_int lda, double * sva, double * u, lapack_int ldu, double * v, lapack_int ldv, double * stat, lapack_int * istat);

lapack_int LAPACKE_cgejsv (int matrix_layout, char joba, char jobu, char jobv, char jobr, char jobt, char jobp, lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, float * sva, lapack_complex_float * u, lapack_int ldu, lapack_complex_float * v, lapack_int ldv, float * stat, lapack_int * istat);

lapack_int LAPACKE_zgejsv (int matrix_layout, char joba, char jobu, char jobv, char jobr, char jobt, char jobp, lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, double * sva, lapack_complex_double * u, lapack_int ldu, lapack_complex_double * v, lapack_int ldv, double * stat, lapack_int * istat);
```
Include Files

- mkl.h

Description

The routine computes the singular value decomposition (SVD) of a real/complex $m$-by-$n$ matrix $A$, where $m \geq n$.

The SVD is written as

$$ A = U \Sigma V^T $$

for real routines

$$ A = U \Sigma V^H $$

for complex routines

where $\Sigma$ is an $m$-by-$n$ matrix which is zero except for its $n$ diagonal elements, $U$ is an $m$-by-$n$ (or $m$-by-$m$) orthonormal matrix, and $V$ is an $n$-by-$n$ orthogonal matrix. The diagonal elements of $\Sigma$ are the singular values of $A$; the columns of $U$ and $V$ are the left and right singular vectors of $A$, respectively. The matrices $U$ and $V$ are computed and stored in the arrays $u$ and $v$, respectively. The diagonal of $\Sigma$ is computed and stored in the array $sva$.

The ?gejsv routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The routine implements a preconditioned Jacobi SVD algorithm. It uses ?geqp3, ?geqrf, and ?gelqf as preprocessors and preconditioners. Optionally, an additional row pivoting can be used as a preprocessor, which in some cases results in much higher accuracy. An example is matrix $A$ with the structure $A = D1 \times C \times D2$, where $D1, D2$ are arbitrarily ill-conditioned diagonal matrices and $C$ is a well-conditioned matrix. In that case, complete pivoting in the first QR factorizations provides accuracy dependent on the condition number of $C$, and independent of $D1, D2$. Such higher accuracy is not completely understood theoretically, but it works well in practice.

If $A$ can be written as $A = B \times D$, with well-conditioned $B$ and some diagonal $D$, then the high accuracy is guaranteed, both theoretically and in software, independent of $D$. For more details see [Drmac08-1], [Drmac08-2].

The computational range for the singular values can be the full range ($\text{UNDERFLOW,OVERFLOW}$), provided that the machine arithmetic and the BLAS and LAPACK routines called by ?gejsv are implemented to work in that range. If that is not the case, the restriction for safe computation with the singular values in the range of normalized IEEE numbers is that the spectral condition number $\kappa(A) = \sigma_{\text{max}}(A)/\sigma_{\text{min}}(A)$ does not overflow. This code (?gejsv) is best used in this restricted range, meaning that singular values of magnitude below $||A||_2 / \text{slamch('O')} \text{ (for single precision) or } ||A||_2 / \text{dlamch('O')} \text{ (for double precision)}$ are returned as zeros. See jobr for details on this.

This implementation is slower than the one described in [Drmac08-1], [Drmac08-2] due to replacement of some non-LAPACK components, and because the choice of some tuning parameters in the iterative part (?gesvj) is left to the implementer on a particular machine.

The rank revealing QR factorization (in this code: ?geqp3) should be implemented as in [Drmac08-3].

If $m$ is much larger than $n$, it is obvious that the initial QRF with column pivoting can be preprocessed by the QRF without pivoting. That well known trick is not used in ?gejsv because in some cases heavy row weighting can be treated with complete pivoting. The overhead in cases $m$ much larger than $n$ is then only due to pivoting, but the benefits in accuracy have prevailed. You can incorporate this extra QRF step easily and also improve data movement (matrix transpose, matrix copy, matrix transposed copy) - this implementation of ?gejsv uses only the simplest, naive data movement.
Input Parameters

\textbf{matrix_layout}

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\textbf{joba}

Must be 'C', 'E', 'F', 'G', 'A', or 'R'.

Specifies the level of accuracy:

If \textit{joba} = 'C', high relative accuracy is achieved if \( A = B \cdot D \) with well-conditioned \( B \) and arbitrary diagonal matrix \( D \). The accuracy cannot be spoiled by column scaling. The accuracy of the computed output depends on the condition of \( B \), and the procedure aims at the best theoretical accuracy. The relative error \( \max_{i=1:N} |d \cdot \sigma_i| / \sigma_i \) is bounded by \( f(M,N) \cdot \text{epsilon} \cdot \text{cond}(B) \), independent of \( D \). The input matrix is preprocessed with the QRF with column pivoting. This initial preprocessing and preconditioning by a rank revealing QR factorization is common for all values of \textit{joba}. Additional actions are specified as follows:

If \textit{joba} = 'E', computation as with 'C' with an additional estimate of the condition number of \( B \). It provides a realistic error bound.

If \textit{joba} = 'F', accuracy higher than in the 'C' option is achieved, if \( A = D_1 \cdot C \cdot D_2 \) with ill-conditioned diagonal scalings \( D_1, D_2 \), and a well-conditioned matrix \( C \). This option is advisable, if the structure of the input matrix is not known and relative accuracy is desirable. The input matrix \( A \) is preprocessed with QR factorization with full (row and column) pivoting.

If \textit{joba} = 'G', computation as with 'F' with an additional estimate of the condition number of \( B \), where \( A = B \cdot D \). If \( A \) has heavily weighted rows, using this condition number gives too pessimistic error bound.

If \textit{joba} = 'A', small singular values are the noise and the matrix is treated as numerically rank deficient. The error in the computed singular values is bounded by \( f(m,n) \cdot \text{epsilon} \cdot ||A|| \). The computed SVD \( A = U \cdot S \cdot V^\top \) (for real flavors) or \( A = U \cdot S \cdot V^\ast \) (for complex flavors) restores \( A \) up to \( f(m,n) \cdot \text{epsilon} \cdot ||A|| \). This enables the procedure to set all singular values below \( \text{epsilon} \cdot ||A|| \) to zero.

If \textit{joba} = 'R', the procedure is similar to the 'A' option. Rank revealing property of the initial QR factorization is used to reveal (using triangular factor) a gap \( \sigma_{r+1} < \text{epsilon} \cdot \sigma_r \), in which case the numerical rank is declared to be \( r \). The SVD is computed with absolute error bounds, but more accurately than with 'A'.
jobu

Must be 'U', 'F', 'W', or 'N'.

Specifies whether to compute the columns of the matrix $U$:

If $jobu = 'U'$, $n$ columns of $U$ are returned in the array $u$.

If $jobu = 'F'$, a full set of $m$ left singular vectors is returned in the array $u$.

If $jobu = 'W'$, $u$ may be used as workspace of length $m \times n$. See the description of $u$.

If $jobu = 'N'$, $u$ is not computed.

jobv

Must be 'V', 'J', 'W', or 'N'.

Specifies whether to compute the matrix $V$:

If $jobv = 'V'$, $n$ columns of $V$ are returned in the array $v$; Jacobi rotations are not explicitly accumulated.

If $jobv = 'J'$, $n$ columns of $V$ are returned in the array $v$ but they are computed as the product of Jacobi rotations. This option is allowed only if $jobu \neq 'N'$.

If $jobv = 'W'$, $v$ may be used as workspace of length $n \times n$. See the description of $v$.

If $jobv = 'N'$, $v$ is not computed.

jobr

Must be 'N' or 'R'.

Specifies the range for the singular values. If small positive singular values are outside the specified range, they may be set to zero. If $A$ is scaled so that the largest singular value of the scaled matrix is around $\sqrt{\text{big}}$, $\text{big} = \text{lamch('O')}$, the function can remove columns of $A$ whose norm in the scaled matrix is less than $\sqrt{\text{lamch('S')}}$ (for $jobr = 'R'$), or less than $\text{small} = \text{lamch('S')}/\text{lamch('E')}$.

If $jobr = 'N'$, the function does not remove small columns of the scaled matrix. This option assumes that BLAS and QR factorizations and triangular solvers are implemented to work in that range. If the condition of $A$ is greater than $\text{big}$, use $\text{gesvj}$.

If $jobr = 'R'$, restricted range for singular values of the scaled matrix $A$ is $[\sqrt{\text{lamch('S')}}, \sqrt{\text{big}}]$, roughly as described above. This option is recommended.

For computing the singular values in the full range $[\text{lamch('S')}, \text{big}]$, use $\text{gesvj}$.

jobt

Must be 'T' or 'N'.

If the matrix is square, the procedure may determine to use a transposed $A$ if $A^T$ (for real flavors) or $A^H$ (for complex flavors) seems to be better with respect to convergence. If the matrix is not square, $jobt$ is ignored.

The decision is based on two values of entropy over the adjoint orbit of $A^T * A$ (for real flavors) or $A^H * A$ (for complex flavors). See the descriptions of $\text{stat}[5]$ and $\text{stat}[6]$. 
If \( jobt = 'T' \), the function performs transposition if the entropy test indicates possibly faster convergence of the Jacobi process, if \( A \) is taken as input. If \( A \) is replaced with \( A^T \) or \( A^H \), the row pivoting is included automatically.

If \( jobt = 'N' \), the functions attempts no speculations. This option can be used to compute only the singular values, or the full SVD \((u, sigma, and v)\). For only one set of singular vectors \((u or v)\), the caller should provide both \( u \) and \( v \), as one of the arrays is used as workspace if the matrix \( A \) is transposed. The implementer can easily remove this constraint and make the code more complicated. See the descriptions of \( u \) and \( v \).

**Caution**
The \( jobt = 'T' \) option is experimental and its effect might not be the same in subsequent releases. Consider using the \( jobt = 'N' \) instead.

\[ \text{jobp} \]

Must be \( 'P' \) or \( 'N' \).

Enables structured perturbations of denormalized numbers. This option should be active if the denormals are poorly implemented, causing slow computation, especially in cases of fast convergence. For details, see [Drmac08-1], [Drmac08-2]. For simplicity, such perturbations are included only when the full SVD or only the singular values are requested. You can add the perturbation for the cases of computing one set of singular vectors.

If \( jobp = 'P' \), the function introduces perturbation.

If \( jobp = 'N' \), the function introduces no perturbation.

\[ m \]

The number of rows of the input matrix \( A \); \( m \geq 0 \).

\[ n \]

The number of columns in the input matrix \( A \); \( m \geq n \geq 0 \).

\[ a, u, v \]

Array \( a(\text{size } lda \times n \text{ for column major layout and } lda \times m \text{ for row major layout}) \) is an array containing the \( m \)-by-\( n \) matrix \( A \).

\( u \) is a workspace array, its size for column major layout is \( ldu \times n \) for \( jobu = 'U' \) or \( 'W' \) and \( ldu \times m \) for \( jobu = 'F' \); for row major layout its size is at least \( ldu \times m \). When \( jobt = 'T' \) and \( m = n \), \( u \) must be provided even though \( jobu = 'N' \).

\( v \) is a workspace array, its size is \( ldv \times n \). When \( jobt = 'T' \) and \( m = n \), \( v \) must be provided even though \( jobv = 'N' \).

\[ lda \]

The leading dimension of the array \( a \). Must be at least \( \max(1, m) \) for column major layout and at least \( \max(1, n) \) for row major layout.

\[ sva \]

\( sva \) is a workspace array, its size is \( n \).

\[ ldu \]

The leading dimension of the array \( u \); \( ldu \geq 1 \).

\[ jobu = 'U' \text{ or 'F' or 'W', } ldu \geq m \text{ for column major layout; for row major layout if } jobu = 'U' \text{ or } jobu = 'W', ldu \geq n \text{ and if } jobu = 'F', ldu \geq m \).

\[ ldv \]

The leading dimension of the array \( v \); \( ldv \geq 1 \).
jobv = 'V' or 'J' or 'W', ldv≥n.

cwork is a workspace array of size \( \max(2, \text{lwork}) \).

rwork is an array of size at least \( \max(7, \text{lrwork}) \) for real flavors and at least \( \max(7, \text{lwork}) \) for complex flavors.

Output Parameters

sva

On exit:

For \( \text{stat}[0]/\text{stat}[1] = 1 \): the singular values of \( A \). During the computation \( \text{sva} \) contains Euclidean column norms of the iterated matrices in the array \( a \).

For \( \text{stat}[0]\ne\text{stat}[1] \): the singular values of \( A \) are \((\text{stat}[0]/\text{stat}[1]) * \text{sva}[0:n-1]\). This factored form is used if \( \text{sigma}_\text{max}(A) \) overflows or if small singular values have been saved from underflow by scaling the input matrix \( A \).

jobr = 'R', some of the singular values may be returned as exact zeros obtained by 'setting to zero' because they are below the numerical rank threshold or are denormalized numbers.

u

On exit:

If \( \text{jobu} = 'U' \), contains the \( m \)-by-\( n \) matrix of the left singular vectors.

If \( \text{jobu} = 'F' \), contains the \( m \)-by-\( m \) matrix of the left singular vectors, including an orthonormal basis of the orthogonal complement of the range of \( A \).

If \( \text{jobu} = 'W' \) and \( \text{jobv} = 'V' \), \( \text{jobt} = 'T' \), and \( m = n \), then \( u \) is used as workspace if the procedure replaces \( A \) with \( A^T \) (for real flavors) or \( A^H \) (for complex flavors). In that case, \( v \) is computed in \( u \) as left singular vectors of \( A^T \) or \( A^H \) and copied back to the \( v \) array. This 'W' option is just a reminder to the caller that in this case \( u \) is reserved as workspace of length \( n*n \).

If \( \text{jobu} = 'N' \), \( u \) is not referenced.

v

On exit:

If \( \text{jobv} = 'V' \) or 'J', contains the \( n \)-by-\( n \) matrix of the right singular vectors.

If \( \text{jobv} = 'W' \) and \( \text{jobu} = 'U' \), \( \text{jobt} = 'T' \), and \( m = n \), then \( v \) is used as workspace if the procedure replaces \( A \) with \( A^T \) (for real flavors) or \( A^H \) (for complex flavors). In that case, \( u \) is computed in \( v \) as right singular vectors of \( A^T \) or \( A^H \) and copied back to the \( u \) array. This 'W' option is just a reminder to the caller that in this case \( v \) is reserved as workspace of length \( n*n \).

If \( \text{jobv} = 'N' \), \( v \) is not referenced.

stat

On exit,

\( \text{stat}[0] = \text{scale} = \text{stat}[1]/\text{stat}[0] \) is the scaling factor such that \( \text{scale}*\text{sva}(1:n) \) are the computed singular values of \( A \). See the description of \( \text{sva} \).
stat[1] = see the description of stat[0].

stat[2] = *sconda* is an estimate for the condition number of column equilibrated A. If joba = 'E' or 'G', *sconda* is an estimate of $\sqrt{|| (R^T \ast R)^{-1} ||_1}$. It is computed using ?pocon. It holds $n^{-1/4} \ast sconda \leq || R^{-1} ||_2 \leq n^{-1/4} + sconda$, where R is the triangular factor from the QRF of A. However, if R is truncated and the numerical rank is determined to be strictly smaller than n, *sconda* is returned as -1, indicating that the smallest singular values might be lost.

If full SVD is needed, the following two condition numbers are useful for the analysis of the algorithm. They are provided for a user who is familiar with the details of the method.

stat[3] = an estimate of the scaled condition number of the triangular factor in the first QR factorization.

stat[4] = an estimate of the scaled condition number of the triangular factor in the second QR factorization.

The following two parameters are computed if jobt = 'T'. They are provided for a user who is familiar with the details of the method.

stat[5] = the entropy of $A^T \ast A$ :: this is the Shannon entropy of $\text{diag}(A^T \ast A) / \text{Trace}(A^T \ast A)$ taken as point in the probability simplex.

stat[6] = the entropy of $A \ast A^{**}t$.

*istat*

On exit,

*istat[0] =* the numerical rank determined after the initial QR factorization with pivoting. See the descriptions of joba and jobr.

*istat[1] =* the number of the computed nonzero singular value.

*istat[2] =* if nonzero, a warning message. If *istat[2] = 1*, some of the column norms of A were denormalized floats. The requested high accuracy is not warranted by the data.

For complex flavors, *istat[3] = 1* or -1. If *istat[3] = 1*, then the procedure used $A^H$ to do the job as specified by the *job* parameters.

**Return Values**

This function returns a value *info*.

If *info=0*, the execution is successful.

If *info = -i*, the *i*-th parameter had an illegal value.

If *info > 0*, the function did not converge in the maximal number of sweeps. The computed values may be inaccurate.

**See Also**

?geqp3

?geqrf

?gelqf

?gesvj

?lamch

?pocon
Computes the singular value decomposition of a real matrix using Jacobi plane rotations.

**Syntax**

```c
lapack_int LAPACKE_sgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, float * a, lapack_int lda, float * sva, lapack_int mv, float * v, lapack_int ldv, float * stat);

lapack_int LAPACKE_dgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, double * a, lapack_int lda, double * sva, lapack_int mv, double * v, lapack_int ldv, double * stat);

lapack_int LAPACKE_cgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, float * sva, lapack_int mv, lapack_complex_float * v, lapack_int ldv, float * stat);

lapack_int LAPACKE_zgesvj (int matrix_layout, char joba, char jobu, char jobv,
lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, double * sva, lapack_int mv, lapack_complex_double * v, lapack_int ldv, double * stat);
```

**Include Files**

- mkl.h

**Description**

The routine computes the singular value decomposition (SVD) of a real or complex $m$-by-$n$ matrix $A$, where $m \geq n$.

The SVD of $A$ is written as

$$ A = U \Sigma V^T $$ for real flavors, or

$$ A = U \Sigma V^H $$ for complex flavors,

where $\Sigma$ is an $m$-by-$n$ diagonal matrix, $U$ is an $m$-by-$n$ orthonormal matrix, and $V$ is an $n$-by-$n$ orthogonal/unitary matrix. The diagonal elements of $\Sigma$ are the singular values of $A$; the columns of $U$ and $V$ are the left and right singular vectors of $A$, respectively. The matrices $U$ and $V$ are computed and stored in the arrays $u$ and $v$, respectively. The diagonal of $\Sigma$ is computed and stored in the array $sva$.

The `gesvj` routine can sometimes compute tiny singular values and their singular vectors much more accurately than other SVD routines.

The $n$-by-$n$ orthogonal matrix $V$ is obtained as a product of Jacobi plane rotations. The rotations are implemented as fast scaled rotations of Anda and Park [AndaPark94]. In the case of underflow of the Jacobi angle, a modified Jacobi transformation of Drmac ([Drmac08-4]) is used. Pivot strategy uses column interchanges of de Rijk ([deRijk98]). The relative accuracy of the computed singular values and the accuracy of the computed singular vectors (in angle metric) is as guaranteed by the theory of Demmel and Veselic [Demmel92]. The condition number that determines the accuracy in the full rank case is essentially

$$ (\min_i d_i) \cdot \kappa(A \cdot D) $$

where $\kappa(.)$ is the spectral condition number. The best performance of this Jacobi SVD procedure is achieved if used in an accelerated version of Drmac and Veselic [Drmac08-1], [Drmac08-2].
The computational range for the nonzero singular values is the machine number interval \((\text{UNDERFLOW}, \text{OVERFLOW})\). In extreme cases, even denormalized singular values can be computed with the corresponding gradual loss of accurate digit.

**Input Parameters**

**matrix_layout**

Specifies whether matrix storage layout is row major (LAPACK\_ROW\_MAJOR) or column major (LAPACK\_COL\_MAJOR).

**joba**

Must be 'L', 'U' or 'G'.

Specifies the structure of A:

If joba = 'L', the input matrix A is lower triangular.

If joba = 'U', the input matrix A is upper triangular.

If joba = 'G', the input matrix A is a general \(m\)-by-\(n\), \(m \geq n\).

**jobu**

Must be 'U', 'C' or 'N'.

Specifies whether to compute the left singular vectors (columns of \(U\)):

If jobu = 'U', the left singular vectors corresponding to the nonzero singular values are computed and returned in the leading columns of A. See more details in the description of \(a\). The default numerical orthogonality threshold is set to approximately \(\text{TOL}=\text{CTOL} \times \text{EPS}\), \(\text{CTOL} = \sqrt{m}\), \(\text{EPS} = \text{?lamch('E')}\)

If jobu = 'C', analogous to jobu = 'U', except that you can control the level of numerical orthogonality of the computed left singular vectors. TOL can be set to \(\text{TOL} = \text{CTOL} \times \text{EPS}\), where \text{CTOL} is given on input in the array \text{stat}. No \text{CTOL} smaller than ONE is allowed. \text{CTOL} greater than \(1 / \text{EPS}\) is meaningless. The option 'C' can be used if \(m \times \text{EPS}\) is satisfactory orthogonality of the computed left singular vectors, so \text{CTOL} = m could save a few sweeps of Jacobi rotations. See the descriptions of \(a\) and \text{stat}[0].

If jobu = 'N', \(u\) is not computed. However, see the description of \(a\).

**jobv**

Must be 'V', 'A' or 'N'.

Specifies whether to compute the right singular vectors, that is, the matrix \(V\):

If jobv = 'V', the matrix \(V\) is computed and returned in the array \(v\).

If jobv = 'A', the Jacobi rotations are applied to the \(mv\)-by-\(n\) array \(v\). In other words, the right singular vector matrix \(V\) is not computed explicitly, instead it is applied to an \(mv\)-by-\(n\) matrix initially stored in the first \(mv\) rows of \(V\).

If jobv = 'N', the matrix \(V\) is not computed and the array \(v\) is not referenced.

**m**

The number of rows of the input matrix A.

\(1 / \text{lamch('E')} > m \geq 0\) for sgsvj.

\(1 / \text{dlamch('E')} > m \geq 0\) for dgesvj.

**n**

The number of columns in the input matrix A; \(m \geq n \geq 0\).
Array $a$ (size at least $lda*n$ for column major layout and $lda*m$ for row major layout) is an array containing the $m$-by-$n$ matrix $A$.

Array $v$ (size at least $\max(1, ldv*n)$) contains, if $jobv = 'A$' the $mv$-by-$n$ matrix to be post-multiplied by Jacobi rotations.

The leading dimension of the array $a$. Must be at least $\max(1, m)$ for column major layout and at least $\max(1, n)$ for row major layout.

If $jobv = 'A$', the product of Jacobi rotations in $?gesvj$ is applied to the first $mv$ rows of $v$. See the description of $jobv$. $0 \leq mv \leq ldv$.

The leading dimension of the array $v$; $ldv \geq 1$.

$jobv = 'V$', $ldv \geq \max(1, n)$.

$jobv = 'A$', $ldv \geq \max(1, mv)$ for column major layout and $ldv \geq \max(1, n)$ for row major layout.

Array size 6. If $jobu = 'C$', $stat[0] = CTOL$, where $CTOL$ defines the threshold for convergence. The process stops if all columns of $A$ are mutually orthogonal up to $CTOL*EPS$, where $EPS = \?lamch('E')$. It is required that $CTOL \geq 1 -$ that is, it is not allowed to force the routine to obtain orthogonality below $\varepsilon$.

Output Parameters

On exit:

If $jobu = 'U$' or $jobu = 'C$':

- if $info = 0$, the leading columns of $A$ contain left singular vectors corresponding to the computed singular values of $a$ that are above the underflow threshold $\?lamch('S')$, that is, non-zero singular values. The number of the computed non-zero singular values is returned in $stat[1]$. Also see the descriptions of $sva$ and $stat$. The computed columns of $u$ are mutually numerically orthogonal up to approximately $TOL = \sqrt{m} \cdot EPS$ (default); or $TOL = CTOL \cdot EPS$ $jobu = 'C$, see the description of $jobu$.

- if $info > 0$, the procedure $?gesvj$ did not converge in the given number of iterations (sweeps). In that case, the computed columns of $u$ may not be orthogonal up to $TOL$. The output $u$ (stored in $a$), $sigma$ (given by the computed singular values in $sva(1:n)$) and $v$ is still a decomposition of the input matrix $A$ in the sense that the residual $||A-scale*U*sigma*V^T||_2 / ||A||_2$ for real flavors or $||A-scale*U*sigma*V^H||_2 / ||A||_2$ for complex flavors (where $scale = stat[0]$) is small.

If $jobu = 'N$':

- if $info = 0$, note that the left singular vectors are 'for free' in the one-sided Jacobi SVD algorithm. However, if only the singular values are needed, the level of numerical orthogonality of $u$ is not an issue and iterations are stopped when the columns of the iterated matrix are numerically orthogonal up to approximately $m \cdot EPS$. Thus, on exit, $a$ contains the columns of $u$ scaled with the corresponding singular values.
• if info > 0, the procedure ?gesvj did not converge in the given number of iterations (sweeps).

**sva**

Array size n.

If info = 0, depending on the value scale = stat[0], where scale is the scaling factor:

• if scale = 1, sva[0:n - 1] contains the computed singular values of a.
• if scale ≠ 1, the singular values of a are scale*sva(1:n), and this factored representation is due to the fact that some of the singular values of a might underflow or overflow.

If info > 0, the procedure ?gesvj did not converge in the given number of iterations (sweeps) and scale*sva(1:n) may not be accurate.

**v**

On exit:

If jobv = 'V', contains the n-by-n matrix of the right singular vectors.

If jobv = 'A', then v contains the product of the computed right singular vector matrix and the initial matrix in the array v.

If jobv = 'N', v is not referenced.

**stat**

On exit,

stat[0] = scale is the scaling factor such that scale*sva(1:n) are the computed singular values of A. See the description of sva.

stat[1] is the number of the computed nonzero singular values.

stat[2] is the number of the computed singular values that are larger than the underflow threshold.

stat[3] is the number of sweeps of Jacobi rotations needed for numerical convergence.

stat[4] = \max_{i ≠ j} |\cos(A(:,i),A(:,j))| in the last sweep. This is useful information in cases when ?gesvj did not converge, as it can be used to estimate whether the output is still useful and for post festum analysis.

stat[5] is the largest absolute value over all sines of the Jacobi rotation angles in the last sweep. It can be useful in a post festum analysis.

**Return Values**

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info > 0, the function did not converge in the maximal number (30) of sweeps. The output may still be useful. See the description of stat.
?gsvd
Computes the generalized singular value decomposition of a pair of general rectangular matrices (deprecated).

Syntax

```c
lapack_int LAPACKE_sggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, float* a,
lapack_int lda, float* b, lapack_int ldb, float* alpha, float* beta, float* u,
lapack_int ldu, float* v, lapack_int ldv, float* q, lapack_int ldq, lapack_int* iwork );
```

```c
lapack_int LAPACKE_dggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l, double* a,
lapack_int lda, double* b, lapack_int ldb, double* alpha, double* beta, double* u,
lapack_int ldu, double* v, lapack_int ldv, double* q, lapack_int ldq, lapack_int* iwork );
```

```c
lapack_int LAPACKE_cggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l,
lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb,
float* alpha, float* beta, lapack_complex_float* u, lapack_int ldu,
lapack_complex_float* v, lapack_int ldv, lapack_complex_float* q, lapack_int ldq,
lapack_int* iwork );
```

```c
lapack_int LAPACKE_zggsvd( int matrix_layout, char jobu, char jobv, char jobq,
lapack_int m, lapack_int n, lapack_int p, lapack_int* k, lapack_int* l,
lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb,
double* alpha, double* beta, lapack_complex_double* u, lapack_int ldu,
lapack_complex_double* v, lapack_int ldv, lapack_complex_double* q, lapack_int ldq,
lapack_int* iwork );
```

Include Files

- mkl.h

Description

This routine is deprecated; use ggsvd3.

The routine computes the generalized singular value decomposition (GSVD) of an \( m \)-by-\( n \) real/complex matrix \( A \) and \( p \)-by-\( n \) real/complex matrix \( B \):

\[
U^*A*Q = D_1*(0 \ R), \quad V^*B*Q = D_2*(0 \ R),
\]

where \( U, V \) and \( Q \) are orthogonal/unitary matrices and \( U', V' \) mean transpose/conjugate transpose of \( U \) and \( V \) respectively.

Let \( k+l \) be the effective numerical rank of the matrix \((A', B')\), then \( R \) is a \((k+l)\)-by-\((k+l)\) nonsingular upper triangular matrix, \( D_1 \) and \( D_2 \) are \( m\)-by-\((k+l)\) and \( p\)-by-\((k+l)\) "diagonal" matrices and of the following structures, respectively:

If \( m-k-l \geq 0 \),
where

\[ C = \text{diag}(\alpha[k], \ldots, \alpha[k + l - 1]) \]

\[ S = \text{diag}(\beta[k], \ldots, \beta[k + l - 1]) \]

\[ C^2 + S^2 = I \]

Nonzero element \( r_{ij} (1 \leq i \leq n) \) of \( R \) is stored in \( a[(i - 1) + (n - k - l + j - 1)*\text{lda}] \) for column major layout and in \( a[(i - 1)*\text{lda} + (n - k - l + j - 1)] \) for row major layout.

If \( m-k-l < 0 \),

\[ D_1 = \begin{pmatrix} k & m-k & 0 & 0 \\ m-k & 0 & C & 0 \\ 0 & C & 0 \\ 0 & 0 & 0 \end{pmatrix} \]

\[ D_2 = \begin{pmatrix} k & m-k & k+l-m \\ m-k & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \]

\[ D_3 = \begin{pmatrix} k & m-k & k+l-m \\ n-k-l & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \]

\[ (0 \ R) = \begin{pmatrix} k & 0 & R_{11} & R_{12} \\ m-k & 0 & R_{21} & R_{22} \\ k+l-m & 0 & R_{31} & R_{32} \end{pmatrix} \]

where

\[ C = \text{diag}(\alpha[k], \ldots, \alpha(m)) \]

\[ S = \text{diag}(\beta[k], \ldots, \beta(m-1)) \]
On exit, the location of nonzero element \( r_{ij} \) of \( R \) depends on the value of \( i \). For \( i \leq m \) this element is stored in \( a[(i - 1) + (n - k - 1 + j - 1)*lda] \) for column major layout and in \( a[(i - 1)*lda + (n - k - 1 + j - 1)] \) for row major layout. For \( m < i \leq k + l \) it is stored in \( b[(i - k - 1) + (n - k - 1 + j - 1)*ldb] \) for column major layout and in \( b[(i - k - 1)*ldb + (n - k - 1 + j - 1)] \) for row major layout.

The routine computes \( C, S, R \), and optionally the orthogonal/unitary transformation matrices \( U, V \) and \( Q \).

In particular, if \( B \) is an \( n \times n \) nonsingular matrix, then the GSVD of \( A \) and \( B \) implicitly gives the SVD of \( A^*B^{-1} \):

\[
A^*B^{-1} = U^*(D_1*D_2^{-1})*V'.
\]

If \( (A', B')' \) has orthonormal columns, then the GSVD of \( A \) and \( B \) is also equal to the CS decomposition of \( A \) and \( B \). Furthermore, the GSVD can be used to derive the solution of the eigenvalue problem:

\[
A'^*A^*x = \lambda*B'^*B^*x.
\]

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobu**
  Must be 'U' or 'N'.
  - If \( jobu = 'U' \), orthogonal/unitary matrix \( U \) is computed.
  - If \( jobu = 'N' \), \( U \) is not computed.

- **jobv**
  Must be 'V' or 'N'.
  - If \( jobv = 'V' \), orthogonal/unitary matrix \( V \) is computed.
  - If \( jobv = 'N' \), \( V \) is not computed.

- **jobq**
  Must be 'Q' or 'N'.
  - If \( jobq = 'Q' \), orthogonal/unitary matrix \( Q \) is computed.
  - If \( jobq = 'N' \), \( Q \) is not computed.

- **m**
  The number of rows of the matrix \( A \) \((m \geq 0)\).

- **n**
  The number of columns of the matrices \( A \) and \( B \) \((n \geq 0)\).

- **p**
  The number of rows of the matrix \( B \) \((p \geq 0)\).

- **a, b**
  Arrays:
  - \( a \) (size at least \( \max(1, \text{lda}\times n) \)) for column major layout and \( \max(1, \text{lda}\times m) \) for row major layout) contains the \( m \times n \) matrix \( A \).
  - \( b \) (size at least \( \max(1, \text{ldb}\times n) \)) for column major layout and \( \max(1, \text{ldb}\times p) \) for row major layout) contains the \( p \times n \) matrix \( B \).

- **lda**
  The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

- **ldb**
  The leading dimension of \( b \); at least \( \max(1, p) \) for column major layout and \( \max(1, n) \) for row major layout.

- **ldu**
  The leading dimension of the array \( u \).
ldu \geq \max(1, m) \text{ if } jobu = 'U'; \ ldu \geq 1 \text{ otherwise.}

\text{The leading dimension of the array } v .

ldv \geq \max(1, p) \text{ if } jobv = 'V'; \ ldv \geq 1 \text{ otherwise.}

ldq \geq \max(1, n) \text{ if } jobq = 'Q'; \ ldq \geq 1 \text{ otherwise.}

\textbf{Output Parameters}

\textit{k, l}

On exit, \(k\) and \(l\) specify the dimension of the subblocks. The sum \(k+l\) is equal to the effective numerical rank of \((A', B')\).

\textit{a}

On exit, \(a\) contains the triangular matrix \(R\) or part of \(R\).

\textit{b}

On exit, \(b\) contains part of the triangular matrix \(R\) if \(m-k-l < 0\).

\textit{alpha, beta}

Arrays, size at least max(1, \(n\)) each.

Contain the generalized singular value pairs of \(A\) and \(B\):

\[ \alpha(1:k) = 1, \]
\[ \beta(1:k) = 0, \]
\[ \text{and if } m-k-l \geq 0, \]
\[ \alpha(k+1:k+l) = c, \]
\[ \beta(k+1:k+l) = s, \]
\[ \text{or if } m-k-l < 0, \]
\[ \alpha(k+1:m) = c, \alpha(m+1:k+l) = 0 \]
\[ \beta(k+1:m) = s, \beta(m+1:k+l) = 1 \]

and

\[ \alpha(k+l+1:n) = 0 \]
\[ \beta(k+l+1:n) = 0. \]

\textit{u, v, q}

Arrays:

\(u\), size at least max(1, \(ldu*m\)).

If \(jobu = 'U'\), \(u\) contains the \(m\)-by-\(m\) orthogonal/unitary matrix \(U\).

If \(jobu = 'N'\), \(u\) is not referenced.

\(v\), size at least max(1, \(ldv*p\)).

If \(jobv = 'V'\), \(v\) contains the \(p\)-by-\(p\) orthogonal/unitary matrix \(V\).

If \(jobv = 'N'\), \(v\) is not referenced.

\(q\), size at least max(1, \(ldq*n\)).

If \(jobq = 'Q'\), \(q\) contains the \(n\)-by-\(n\) orthogonal/unitary matrix \(Q\).

If \(jobq = 'N'\), \(q\) is not referenced.

\textit{iwork}

On exit, \(iwork\) stores the sorting information.
Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info = 1, the Jacobi-type procedure failed to converge. For further details, see subroutine tgsja.

?gesvd
Computes the SVD and left and right singular vectors for a matrix.

Syntax

lapack_int LAPACKE_sgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, float * a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int iu, lapack_int * ns, float * s, float * u, lapack_int ldu, float * vt, lapack_int ldvt, lapack_int * superb);

lapack_int LAPACKE_dgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, double * a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int iu, lapack_int * ns, double * s, double * u, lapack_int ldu, double * vt, lapack_int ldvt, lapack_int * superb);

lapack_int LAPACKE_cgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, float vl, float vu, lapack_int il, lapack_int iu, lapack_int * ns, float * s, lapack_complex_float * u, lapack_int ldu, lapack_complex_float * vt, lapack_int ldvt, lapack_int * superb);

lapack_int LAPACKE_zgesvdx (int matrix_layout, char jobu, char jobvt, char range,
lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, double vl, double vu, lapack_int il, lapack_int iu, lapack_int * ns, double * s, lapack_complex_double * u, lapack_int ldu, lapack_complex_double * vt, lapack_int ldvt, lapack_int * superb);

Include Files

• mkl.h

Description

?gesvdx computes the singular value decomposition (SVD) of a real or complex m-by-n matrix A, optionally computing the left and right singular vectors. The SVD is written

\[ A = U \times \Sigma \times \text{transpose}(V) \]

where \( \Sigma \) is an \( m \)-by-\( n \) matrix which is zero except for its \( \min(m,n) \) diagonal elements, \( U \) is an \( m \)-by-\( m \) matrix, and \( V \) is an \( n \)-by-\( n \) matrix. The matrices \( U \) and \( V \) are orthogonal for real \( A \), and unitary for complex \( A \). The diagonal elements of \( \Sigma \) are the singular values of \( A \); they are real and non-negative, and are returned in descending order. The first \( \min(m,n) \) columns of \( U \) and \( V \) are the left and right singular vectors of \( A \).

?gesvdx uses an eigenvalue problem for obtaining the SVD, which allows for the computation of a subset of singular values and vectors. See ?bdsvx for details.

Note that the routine returns \( V^T \), not \( V \).
**Input Parameters**

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**jobu**
Specifies options for computing all or part of the matrix $U$:
- 'V': the first min($m,n$) columns of $U$ (the left singular vectors) or as specified by range are returned in the array $u$;
- 'N': no columns of $U$ (no left singular vectors) are computed.

**jobvt**
Specifies options for computing all or part of the matrix $V^T$:
- 'V': the first min($m,n$) rows of $V^T$ (the right singular vectors) or as specified by range are returned in the array $vt$;
- 'N': no rows of $V^T$ (no right singular vectors) are computed.

**range**
- 'A': find all singular values.
- 'V': all singular values in the half-open interval $(vl,vu]$ are found.
- 'I': the $il$-th through $iu$-th singular values are found.

**m**
The number of rows of the input matrix $A$. $m \geq 0$.

**n**
The number of columns of the input matrix $A$. $n \geq 0$.

**a**
Array, size $lda*n$
On entry, the $m$-by-$n$ matrix $A$.

**lda**
The leading dimension of the array $a$.
$lda \geq \max(1,m)$.

**vl**
$vl \geq 0$.

**vu**
If range='V', the lower and upper bounds of the interval to be searched for singular values. $vu > vl$. Not referenced if range = 'A' or 'I'.

**il**

**iu**
If range='I', the indices (in ascending order) of the smallest and largest singular values to be returned. $1 \leq il \leq iu \leq \min(m,n)$, if $\min(m,n) > 0$. Not referenced if range = 'A' or 'V'.

**ldu**
The leading dimension of the array $u$. $ldu \geq 1$; if jobu = 'V', $ldu \geq m$.

**ldvt**
The leading dimension of the array $vt$. $ldvt \geq 1$; if jobvt = 'V', $ldvt \geq ns$ (see above).

**Output Parameters**

**a**
On exit, the contents of $a$ are destroyed.

**ns**
The total number of singular values found,
$0 \leq ns \leq \min(m,n)$.
If range = 'A', $ns = \min(m,n)$; if range = 'I', $ns = iu - il + 1$. 


The singular values of $A$, sorted so that $s[i] \geq s[i + 1]$.

**Return Values**

This function returns a value $info$.

$= 0$: successful exit.

$< 0$: if $info = -i$, the $i$-th argument had an illegal value.

$> 0$: if $info = i$, then $i$ eigenvectors failed to converge in ?bdsdx/?stevx. if $info = n*2 + 1$, an internal error occurred in ?bdsdx.

?bdsdx

Computes the SVD of a bidiagonal matrix.

**Syntax**

```c
lapack_int LAPACKE_sbdsvdx (int matrix_layout, char uplo, char jobz, char range,
lapack_int n, float * d, float * e, float vl, float vu, lapack_int il, lapack_int iu,
lapack_int * ns, float * s, float * z, lapack_int ldz, lapack_int * superb);

lapack_int LAPACKE_dbdsvdx (int matrix_layout, char uplo, char jobz, char range,
lapack_int n, double * d, double * e, double vl, double vu, lapack_int il, lapack_int iu,
lapack_int * ns, double * s, double * z, lapack_int ldz, lapack_int * superb);
```

**Include Files**

- mkl.h
Description

?bdsvdx computes the singular value decomposition (SVD) of a real n-by-n (upper or lower) bidiagonal matrix $B$, $B = U * S * V^T$, where $S$ is a diagonal matrix with non-negative diagonal elements (the singular values of $B$), and $U$ and $V$ are orthogonal matrices of left and right singular vectors, respectively.

Given an upper bidiagonal $B$ with diagonal $d = [d_1d_2 \ldots d_n]$ and superdiagonal $e = [e_1e_2 \ldots e_{n-1}]$, ?bdsvdx computes the singular value decomposition of $B$ through the eigenvalues and eigenvectors of the $n*2$-by-$n*2$ tridiagonal matrix

$$TGK = \begin{pmatrix} 0 & d_1 \\ d_1 & 0 & e_1 \\ & d_2 & \ddots & \vdots \\ & e_1 & \ddots & \ddots \\ \end{pmatrix}$$

If $(s,u,v)$ is a singular triplet of $B$ with $||u|| = ||v|| = 1$, then $(\pm s, q)$, $||q|| = 1$, are eigenpairs of TGK, with

$$q = P * \left( \frac{u' \pm v'}{\sqrt{2}} \right) = \left( \frac{v_1 u_1 u_2 u_3 \cdots v_n u_n}{\sqrt{2}} \right),$$

and

$$P = \left( e_{n+1} e_1 e_{n+2} e_2 \cdots \right).$$

Given a TGK matrix, one can either

1. compute $-s$, $-v$ and change signs so that the singular values (and corresponding vectors) are already in descending order (as in ?gesvd/?gesdd) or
2. compute $s$, $v$ and reorder the values (and corresponding vectors).


Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  = 'U': $B$ is upper bidiagonal;
  = 'L': $B$ is lower bidiagonal.

- **jobz**
  = 'N': Compute singular values only;
  = 'V': Compute singular values and singular vectors.

- **range**
  = 'A': Find all singular values.
  = 'V': all singular values in the half-open interval $[vl,vu]$ are found.
  = 'I': the $il$-th through $iu$-th singular values are found.

- **n**
  The order of the bidiagonal matrix.
  $n \geq 0$.

- **d**
  Array, size $n$.
  The $n$ diagonal elements of the bidiagonal matrix $B$.

- **e**
  Array, size $(\text{max}(1,n-1))$
  The $(n-1)$ superdiagonal elements of the bidiagonal matrix $B$ in elements 1 to $n-1$. 
If range='V', the lower and upper bounds of the interval to be searched for singular values. vu > vl.
Not referenced if range = 'A' or 'I'.
If range='I', the indices (in ascending order) of the smallest and largest singular values to be returned.
1 ≤ il ≤ iu ≤ min(m,n), if min(m,n) > 0.
Not referenced if range = 'A' or 'V'.
The leading dimension of the array z.
ldz ≥ 1, and if jobz = 'V', ldz ≥ max(2,n*2).

Output Parameters

ns
The total number of singular values found. 0 ≤ ns ≤ n.
If range = 'A', ns = n, and if range = 'I', ns = iu - il + 1.
s
Array, size (n)
The first ns elements contain the selected singular values in ascending order.
z
Array, size 2*n*k
If jobz = 'V', then if info = 0 the first ns columns of z contain the singular vectors of the matrix B corresponding to the selected singular values, with U in rows 1 to n and V in rows n+1 to n*2, i.e.
\[ z = \begin{bmatrix} U \\ V \end{bmatrix} \]
If jobz = 'N', then z is not referenced.

NOTE
Make sure that at least k = ns+1 columns are supplied in the array z; if range = 'V', the exact value of ns is not known in advance and an upper bound must be used.

superb
Array, size (12*n).
If jobz = 'V', then if info = 0, the first ns elements of iwork are zero. If info > 0, then iwork contains the indices of the eigenvectors that failed to converge in ?stevx.

Return Values
This function returns a value info.
= 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.
> 0:
if info = i, then i eigenvectors failed to converge in ?stevx. The indices of the eigenvectors (as returned by ?stevx) are stored in the array iwork.

if info = n*2 + 1, an internal error occurred.

**Cosine-Sine Decomposition: LAPACK Driver Routines**

This section describes LAPACK driver routines for computing the cosine-sine decomposition (CS decomposition). You can also call the corresponding computational routines to perform the same task.

The computation has the following phases:

1. The matrix is reduced to a bidiagonal block form.
2. The blocks are simultaneously diagonalized using techniques from the bidiagonal SVD algorithms.

Table "Driver Routines for Cosine-Sine Decomposition (CSD)" lists LAPACK routines that perform CS decomposition of matrices.

**Computational Routines for Cosine-Sine Decomposition (CSD)**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real matrices</th>
<th>Complex matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute the CS decomposition of a block-partitioned orthogonal matrix</td>
<td>orcsd uncsd</td>
<td>orcsd uncsd</td>
</tr>
<tr>
<td>Compute the CS decomposition of a block-partitioned unitary matrix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

CS Computational Routines

?orcsd/?uncsd

*Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.*

**Syntax**

```c
lapack_int LAPACKE_sorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, float* x11, lapack_int ldx11, float* x12, lapack_int ldx12, float* x21, lapack_int ldx21, float* x22, lapack_int ldx22, float* theta, float* u1, lapack_int ldu1, float* u2, lapack_int ldu2, float* vt, lapack_int ldvt1, float* v2t, lapack_int ldv2t );
```

```c
lapack_int LAPACKE_dorcsd( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, double* x11, lapack_int ldx11, double* x12, lapack_int ldx12, double* x21, lapack_int ldx21, double* x22, lapack_int ldx22, double* theta, double* u1, lapack_int ldu1, double* u2, lapack_int ldu2, double* vt, lapack_int ldvt1, double* v2t, lapack_int ldv2t );
```

```c
lapack_int LAPACKE_cunge( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, lapack_complex_float* x11, lapack_int ldx11, lapack_complex_float* x12, lapack_int ldx12, lapack_complex_float* x21, lapack_int ldx21, lapack_complex_float* x22, lapack_int ldx22, lapack_complex_float* theta, lapack_complex_float* u1, lapack_int ldu1, lapack_complex_float* u2, lapack_int ldu2, lapack_complex_float* vt, lapack_int ldvt1, lapack_complex_float* v2t, lapack_int ldv2t );
```

```c
lapack_int LAPACKE_zunge( int matrix_layout, char jobu1, char jobu2, char jobv1t, char jobv2t, char trans, char signs, lapack_int m, lapack_int p, lapack_int q, lapack_complex_double* x11, lapack_int ldx11, lapack_complex_double* x12, lapack_int ldx12, lapack_complex_double* x21, lapack_int ldx21, lapack_complex_double* x22, lapack_int ldx22, lapack_complex_double* theta, lapack_complex_double* u1, lapack_int ldu1, lapack_complex_double* u2, lapack_int ldu2, lapack_complex_double* vt, lapack_int ldvt1, lapack_complex_double* v2t, lapack_int ldv2t );
```
lapack_int ldx22, double* theta, lapack_complex_double* u1, lapack_int ldu1,
lapack_complex_double* u2, lapack_int ldu2, lapack_complex_double* v1t, lapack_int
ldv1t, lapack_complex_double* v2t, lapack_int ldv2t);

Include Files
• mkl.h

Description
The routines ?orcsd/?uncsd compute the CS decomposition of an \(m\)-by-\(m\) partitioned orthogonal matrix \(X\):

\[
X = \begin{pmatrix}
    x_{11} & x_{12} \\
    x_{21} & x_{22}
\end{pmatrix} =
\begin{pmatrix}
    u_1 \mid u_2
\end{pmatrix}
\begin{pmatrix}
    I & 0 & 0 & 0 & 0 \\
    0 & C & 0 & -S & 0 \\
    0 & 0 & 0 & 0 & -I \\
    0 & 0 & 0 & I & 0 \\
    0 & S & 0 & 0 & C \\
    0 & 0 & I & 0 & 0
\end{pmatrix}
\begin{pmatrix}
    v_1 \mid v_2
\end{pmatrix}
\]

or unitary matrix:

\[
X = \begin{pmatrix}
    x_{11} & x_{12} \\
    x_{21} & x_{22}
\end{pmatrix} =
\begin{pmatrix}
    u_1 \mid u_2
\end{pmatrix}
\begin{pmatrix}
    I & 0 & 0 & 0 & 0 \\
    0 & C & 0 & -S & 0 \\
    0 & 0 & 0 & 0 & -I \\
    0 & 0 & 0 & I & 0 \\
    0 & S & 0 & 0 & C \\
    0 & 0 & I & 0 & 0
\end{pmatrix}
\begin{pmatrix}
    v_1 \mid v_2
\end{pmatrix}
\]

\(x_{11}\) is \(p\)-by-\(q\). The orthogonal/unitary matrices \(u_1, u_2, v_1,\) and \(v_2\) are \(p\)-by-\(p\), \((m-p)\)-by-\((m-p)\), \(q\)-by-\(q\), \((m-q)\)-by-\((m-q)\), respectively. \(C\) and \(S\) are \(r\)-by-\(r\) nonnegative diagonal matrices satisfying \(C^2 + S^2 = I\), in which \(r = \min(p, m-p, q, m-q)\).

Input Parameters

\(\text{matrix\_layout}\)

- Specifies whether matrix storage layout is row major (\text{LAPACK\_ROW\_MAJOR}) or column major (\text{LAPACK\_COL\_MAJOR}).

\(\text{jobu1}\)

- If equals \(Y\), then \(u_1\) is computed. Otherwise, \(u_1\) is not computed.

\(\text{jobu2}\)

- If equals \(Y\), then \(u_2\) is computed. Otherwise, \(u_2\) is not computed.

\(\text{jobv1t}\)

- If equals \(Y\), then \(v_1^t\) is computed. Otherwise, \(v_1^t\) is not computed.

\(\text{jobv2t}\)

- If equals \(Y\), then \(v_2^t\) is computed. Otherwise, \(v_2^t\) is not computed.

\(\text{trans}\)

\(= \text{’T’}: \)

- \(x, u_1, u_2, v_1^t, v_2^t\) are stored in row-major order.

\(\text{otherwise}\)

- \(x, u_1, u_2, v_1^t, v_2^t\) are stored in column-major order.

\(\text{signs}\)

\(= \text{’O’}: \)

- The lower-left block is made nonpositive (the "other" convention).
otherwise The upper-right block is made nonpositive (the "default" convention).

$m$
The number of rows and columns of the matrix $X$.

$p$
The number of rows in $x_{11}$ and $x_{12}$. $0 \leq p \leq m$.

$q$
The number of columns in $x_{11}$ and $x_{21}$. $0 \leq q \leq m$.

$x_{11}, x_{12}, x_{21}, x_{22}$ Arrays of size $x_{11}(ldx_{11}, q)$, $x_{12}(ldx_{12}, m - q)$, $x_{21}(ldx_{21}, q)$, and $x_{22}(ldx_{22}, m - q)$.

Contain the parts of the orthogonal/unitary matrix whose CSD is desired.

$ldx_{11}, ldx_{12}, ldx_{21}, ldx_{22}$ The leading dimensions of the parts of array $X$. $ldx_{11} \geq \max(1, p)$, $ldx_{12} \geq \max(1, q)$, $ldx_{21} \geq \max(1, m - p)$, $ldx_{22} \geq \max(1, m - q)$.

$ldu_1$ The leading dimension of the array $u_1$. If $jobu_1 = 'Y'$, $ldu_1 \geq \max(1, p)$.

$ldu_2$ The leading dimension of the array $u_2$. If $jobu_2 = 'Y'$, $ldu_2 \geq \max(1, m - p)$.

$ldv_1t$ The leading dimension of the array $v_1t$. If $jobv_1t = 'Y'$, $ldv_1t \geq \max(1, q)$.

$ldv_2t$ The leading dimension of the array $v_2t$. If $jobv_2t = 'Y'$, $ldv_2t \geq \max(1, m - q)$.

Output Parameters

$theta$ Array, size $r$, in which $r = \min(p, m-p, q, m-q)$. $C = \text{diag}(\cos(\theta[0]), ..., \cos(\theta[r-1]))$, and $S = \text{diag}(\sin(\theta[0]), ..., \sin(\theta[r-1]))$.

$u_1$ Array, size at least $\max(1, ldu_1*p)$.

If $jobu_1 = 'Y'$, $u_1$ contains the $p$-by-$p$ orthogonal/unitary matrix $u_1$.

$u_2$ Array, size at least $\max(1, ldu_2*(m - p))$.

If $jobu_2 = 'Y'$, $u_2$ contains the $(m-p)$-by-$(m-p)$ orthogonal/unitary matrix $u_2$.

$v_1t$ Array, size at least $\max(1, ldv_1t*q)$.

If $jobv_1t = 'Y'$, $v_1t$ contains the $q$-by-$q$ orthogonal matrix $v_1^T$ or unitary matrix $v_1^H$.

$v_2t$ Array, size at least $\max(1, ldv_2t*(m - q))$.

If $jobv_2t = 'Y'$, $v_2t$ contains the $(m-q)$-by-$(m-q)$ orthogonal matrix $v_2^T$ or unitary matrix $v_2^H$.

Return Values

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.
> 0: ?orcsd/?uncsd did not converge.

See Also
?bbcscd
xerbla

?orcsd2by1/?uncsd2by1
Computes the CS decomposition of a block-partitioned orthogonal/unitary matrix.

Syntax
lapack_int LAPACKE_sorcsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, float * x11, lapack_int ldx11, float * x21,
lapack_int ldx21, float * theta, float * u1, lapack_int ldu1, float * u2, lapack_int
ldu2, float * v1t, lapack_int ldv1t);

lapack_int LAPACKE_dorcsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, double * x11, lapack_int ldx11, double * x21,
lapack_int ldx21, double * theta, double * u1, lapack_int ldu1, double * u2, lapack_int
ldu2, double * v1t, lapack_int ldv1t);

lapack_int LAPACKE_cuncsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, lapack_complex_float * x11, lapack_int ldx11,
lapack_complex_float * x21, lapack_int ldx21, float * theta, lapack_complex_float * u1,
lapack_int ldu1, lapack_complex_float * u2, lapack_int ldu2, lapack_complex_float *
v1t, lapack_int ldv1t);

lapack_int LAPACKE_zuncsd2by1 (int matrix_layout, char jobu1, char jobu2, char jobv1t,
lapack_int m, lapack_int p, lapack_int q, lapack_complex_double * x11, lapack_int ldx11,
lapack_complex_double * x21, lapack_int ldx21, float * theta, lapack_complex_double * u1,
lapack_int ldu1, lapack_complex_double * u2, lapack_int ldu2, lapack_complex_double *
v1t, lapack_int ldv1t);

Include Files
• mkl.h

Description
The routines ?orcsd2by1/?uncsd2by1 compute the CS decomposition of an \( m \)-by-\( q \) matrix \( X \) with orthonormal columns that has been partitioned into a 2-by-1 block structure:

\[
X = \begin{bmatrix}
X_{11} \\
X_{21}
\end{bmatrix}
= \begin{bmatrix}
U_1 & U_2
\end{bmatrix}
\begin{bmatrix}
I & 0 & 0 \\
0 & C & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & S & 0 \\
0 & 0 & I
\end{bmatrix}
\begin{bmatrix}
V_1^\text{H} \\
V_2
\end{bmatrix}
\]

\( x_{11} \) is \( p \)-by-\( q \). The orthogonal/unitary matrices \( u_1, u_2, v_1, \) and \( v_2 \) are \( p \)-by-\( p \), \((m-p)\)-by-\((m-p)\), \( q \)-by-\( q \), \((m-q)\)\( m-q \), respectively. \( C \) and \( S \) are \( r \)-by-\( r \) nonnegative diagonal matrices satisfying \( C^2 + S^2 = I \), in which \( r = \min(p, m-p, q, m-q) \).
Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **jobu1**: If equal to 'Y', then $u_1$ is computed. Otherwise, $u_1$ is not computed.
- **jobu2**: If equal to 'Y', then $u_2$ is computed. Otherwise, $u_2$ is not computed.
- **jobv1t**: If equal to 'Y', then $v_1^T$ is computed. Otherwise, $v_1^T$ is not computed.
- **m**: The number of rows and columns of the matrix $X$.
- **p**: The number of rows in $x_{11}$. $0 \leq p \leq m$.
- **q**: The number of columns in $x_{11}$. $0 \leq q \leq m$.
- **x11**: Array, size $(ldx11*p)$. On entry, the part of the orthogonal matrix whose CSD is desired.
- **ldx11**: The leading dimension of the array $x_{11}$. $ldx11 \geq \max(1, p)$.
- **x21**: Array, size $(ldx21*q)$. On entry, the part of the orthogonal matrix whose CSD is desired.
- **ldx21**: The leading dimension of the array $X$. $ldx21 \geq \max(1, m-p)$.
- **ldu1**: The leading dimension of the array $u_1$. If $jobu1 = 'Y'$, $ldu1 \geq \max(1, p)$.
- **ldu2**: The leading dimension of the array $u_2$. If $jobu2 = 'Y'$, $ldu2 \geq \max(1, m-p)$.
- **ldv1t**: The leading dimension of the array $v1t$. If $jobv1t = 'Y'$, $ldv1t \geq \max(1, q)$.

Output Parameters

- **theta**: Array, size $r$, in which $r = \min(p, m-p, q, m-q)$. 
  
  $C = \text{diag} \left( \cos(\theta(1)), \ldots, \cos(\theta(r)) \right)$, and 
  
  $S = \text{diag} \left( \sin(\theta(1)), \ldots, \sin(\theta(r)) \right)$.
- **u1**: Array, size $(ldu1*p)$.
  
  If $jobu1 = 'Y'$, $u1$ contains the $p$-by-$p$ orthogonal/unitary matrix $u_1$.
- **u2**: Array, size $(ldu2*(m-p))$.
  
  If $jobu2 = 'Y'$, $u2$ contains the $(m-p)$-by-$(m-p)$ orthogonal/unitary matrix $u_2$.
- **v1t**: Array, size $(ldv1t*q)$.
  
  If $jobv1t = 'Y'$, $v1t$ contains the $q$-by-$q$ orthogonal matrix $v_1^T$ or unitary matrix $v_1^H$.

Return Values

This function returns a value info.

- $= 0$: successful exit
< 0: if info = -i, the i-th argument has an illegal value
> 0: ?orcsd2by1/?uncsd2by1 did not converge.

See Also
?bbcsd
xerbla

Generalized Symmetric Definite Eigenvalue Problems: LAPACK Driver Routines

This section describes LAPACK driver routines used for solving generalized symmetric definite eigenproblems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Generalized Symmetric Definite Eigenproblems" lists all such driver routines.

Driver Routines for Solving Generalized Symmetric Definite Eigenproblems

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>sygv/hegv</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.</td>
</tr>
<tr>
<td>sygvd/hegvd</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem. If eigenvectors are desired, it uses a divide and conquer method.</td>
</tr>
<tr>
<td>sygvx/hegx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem.</td>
</tr>
<tr>
<td>spgv/hpgv</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.</td>
</tr>
<tr>
<td>spgvd/hpgvd</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage. If eigenvectors are desired, it uses a divide and conquer method.</td>
</tr>
<tr>
<td>spgvx/hpgvx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with matrices in packed storage.</td>
</tr>
<tr>
<td>sbgv/hbgv</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.</td>
</tr>
<tr>
<td>sbgvd/hbgvd</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.</td>
</tr>
<tr>
<td>sbgvx/hbgvx</td>
<td>Computes selected eigenvalues and, optionally, eigenvectors of a real / complex generalized symmetric /Hermitian positive-definite eigenproblem with banded matrices.</td>
</tr>
</tbody>
</table>

sygv

*Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.*

Syntax

```c
lapack_int LAPACKE_ssygv (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* w);

lapack_int LAPACKE_dsygv (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* w);
```
Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[ A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x. \]

Here \( A \) and \( B \) are assumed to be symmetric and \( B \) is also positive definite.

Input Parameters

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **itype**
  
  Must be 1 or 2 or 3.

  Specifies the problem type to be solved:

  - if \( itype = 1 \), the problem type is \( A^*x = \lambda^*B^*x \);
  - if \( itype = 2 \), the problem type is \( A^*B^*x = \lambda^*x \);
  - if \( itype = 3 \), the problem type is \( B^*A^*x = \lambda^*x \).

- **jobz**
  
  Must be 'N' or 'V'.

  If \( jobz = 'N' \), then compute eigenvalues only.

  If \( jobz = 'V' \), then compute eigenvalues and eigenvectors.

- **uplo**
  
  Must be 'U' or 'L'.

  If \( uplo = 'U' \), arrays \( a \) and \( b \) store the upper triangles of \( A \) and \( B \);

  If \( uplo = 'L' \), arrays \( a \) and \( b \) store the lower triangles of \( A \) and \( B \).

- **n**
  
  The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

- **a, b**
  
  Arrays:

  - \( a \) (size at least max(1, lda*n)) contains the upper or lower triangle of the symmetric matrix \( A \), as specified by \( uplo \).
  
  - \( b \) (size at least max(1, ldb*n)) contains the upper or lower triangle of the symmetric positive definite matrix \( B \), as specified by \( uplo \).

- **lda**
  
  The leading dimension of \( a \); at least max(1, n).

- **ldb**
  
  The leading dimension of \( b \); at least max(1, n).

Output Parameters

- **a**
  
  On exit, if \( jobz = 'V' \), then if \( info = 0 \), \( a \) contains the matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:

  - if \( itype = 1 \) or 2, \( Z^*B^*Z = I \);
  - if \( itype = 3 \), \( Z^*\text{inv}(B)^*Z = I \);
If jobz = 'N', then on exit the upper triangle (if uplo = 'U') or the lower triangle (if uplo = 'L') of A, including the diagonal, is destroyed.

On exit, if info≤n, the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization B = U^T U or B = L*L^T.

Array, size at least max(1, n).
If info = 0, contains the eigenvalues in ascending order.

Return Values
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
If info > 0, spotrf/dpotrf or ssyev/dsyev returned an error code:
If info = i≤n, ssyev/dsyev failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;
If info = n + i, for 1 ≤i≤n, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?hegv
Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax
lapack_int LAPACKE_chegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b,
lapack_int ldb, float* w );
lapack_int LAPACKE_zhegv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, double* w );

Include Files
• mkl.h

Description
The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form
A*x = λ*B*x,  A*B*x = λ*x, or B*A*x = λ*x.

Here A and B are assumed to be Hermitian and B is also positive definite.

Input Parameters
matrix_layout       Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
itype

Must be 1 or 2 or 3. Specifies the problem type to be solved:
if itype = 1, the problem type is $A*x = \lambda*B*x$;
if itype = 2, the problem type is $A*B*x = \lambda*x$;
if itype = 3, the problem type is $B*A*x = \lambda*x$.

jobz

Must be 'N' or 'V'.
If jobz = 'N', then compute eigenvalues only.
If jobz = 'V', then compute eigenvalues and eigenvectors.

uplo

Must be 'U' or 'L'.
If uplo = 'U', arrays $a$ and $b$ store the upper triangles of $A$ and $B$;
If uplo = 'L', arrays $a$ and $b$ store the lower triangles of $A$ and $B$.

n

The order of the matrices $A$ and $B$ ($n \geq 0$).

$a, b$

Arrays:
a (size at least max(1, lda*n)) contains the upper or lower triangle of the
Hermitian matrix $A$, as specified by uplo.
b (size at least max(1, ldb*n)) contains the upper or lower triangle of the
Hermitian positive definite matrix $B$, as specified by uplo.

lda

The leading dimension of $a$; at least max(1, $n$).

ldb

The leading dimension of $b$; at least max(1, $n$).

Output Parameters

$a$

On exit, if jobz = 'V', then if info = 0, $a$ contains the matrix $Z$ of
eigenvectors. The eigenvectors are normalized as follows:
if itype = 1 or 2, $Z^H*B*Z = I$;
if itype = 3, $Z^H*inv(B)*Z = I$;
If jobz = 'N', then on exit the upper triangle (if uplo = 'U') or the
lower triangle (if uplo = 'L') of $A$, including the diagonal, is destroyed.

$b$

On exit, if info<=n, the part of $b$ containing the matrix is overwritten by the
triangular factor $U$ or $L$ from the Cholesky factorization $B = U^H*U$ or $B =
L*L^H$.

w

Array, size at least max(1, $n$).
If info = 0, contains the eigenvalues in ascending order.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info=-i, the i-th parameter had an illegal value.
If info > 0, cpotrf/zpotrf or cheev/zheev return an error code:
If $info = i\leq n$, cheev/zheev fails to converge, and $i$ off-diagonal elements of an intermediate tridiagonal do not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order $i$ of $B$ is not positive-definite. The factorization of $B$ can not be completed and no eigenvalues or eigenvectors are computed.

?sygvd
Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Syntax

```c
lapack_int LAPACKE_ssygvd(int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* w);

lapack_int LAPACKE_dsygvd(int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* w);
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

$$A^*x = \lambda*B^*x, \quad A^*B^*x = \lambda*x, \quad \text{or} \quad B^*A^*x = \lambda*x.$$ 

Here $A$ and $B$ are assumed to be symmetric and $B$ is also positive definite.

It uses a divide and conquer algorithm.

Input Parameters

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **itype**
  - Must be 1, 2, or 3. Specifies the problem type to be solved:
    - If $itype = 1$, the problem type is $A^*x = \lambda B^*x$;
    - If $itype = 2$, the problem type is $A^*B^*x = \lambda x$;
    - If $itype = 3$, the problem type is $B^*A^*x = \lambda x$.

- **jobz**
  - Must be 'N' or 'V'.
    - If $jobz = 'N'$, then compute eigenvalues only.
    - If $jobz = 'V'$, then compute eigenvalues and eigenvectors.

- **uplo**
  - Must be 'U' or 'L'.
    - If $uplo = 'U'$, arrays $a$ and $b$ store the upper triangles of $A$ and $B$;
    - If $uplo = 'L'$, arrays $a$ and $b$ store the lower triangles of $A$ and $B$.

- **n**
  - The order of the matrices $A$ and $B$ ($n \geq 0$).

- **a, b**
  - Arrays:
\( a \) (size at least \( \text{lda}*n \)) contains the upper or lower triangle of the symmetric matrix \( A \), as specified by \( \text{uplo} \).

\( b \) (size at least \( \text{ldb}*n \)) contains the upper or lower triangle of the symmetric positive definite matrix \( B \), as specified by \( \text{uplo} \).

Output Parameters

\( a \)

On exit, if \( \text{jobz} = 'V' \), then if \( \text{info} = 0 \), \( a \) contains the matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:

- If \( \text{itype} = 1 \) or \( 2 \), \( Z^T*B*Z = I \);
- If \( \text{itype} = 3 \), \( Z^T*\text{inv}(B)*Z = I \);

If \( \text{jobz} = 'N' \), then on exit the upper triangle (if \( \text{uplo} = 'U' \)) or the lower triangle (if \( \text{uplo} = 'L' \)) of \( A \), including the diagonal, is destroyed.

\( b \)

On exit, if \( \text{info} \leq n \), the part of \( b \) containing the matrix is overwritten by the triangular factor \( U \) or \( L \) from the Cholesky factorization \( B = U^T*U \) or \( B = L*L^T \).

\( w \)

Array, size at least \( \text{max}(1, n) \).

If \( \text{info} = 0 \), contains the eigenvalues in ascending order.

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} > 0 \), an error code is returned as specified below.

- For \( \text{info} \leq n \):
  - If \( \text{info} = i \) and \( \text{jobz} = 'N' \), then the algorithm failed to converge; \( i \) off-diagonal elements of an intermediate tridiagonal form did not converge to zero.
  - If \( \text{jobz} = 'V' \), then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns \( \text{info}/(n+1) \) through \( \text{mod}(\text{info},n+1) \).

- For \( \text{info} > n \):
  - If \( \text{info} = n + i \), for \( 1 \leq i \leq n \), then the leading minor of order \( i \) of \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

?hegvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem using a divide and conquer method.

Syntax

\[
\text{lapack_int LAPACKE_chegvd( int matrix_layout, int itype, char jobz, char uplo, int n, lapack_complex_float* a, int lda, lapack_complex_float* b, int ldb, float* w );}
\]
lapack_int LAPACKE_zhegvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b,
lapack_int ldb, double* w );

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

\[ A\mathbf{x} = \lambda B\mathbf{x}, \quad A^*B\mathbf{x} = \lambda\mathbf{x}, \text{ or } B^*A\mathbf{x} = \lambda\mathbf{x}. \]

Here \( A \) and \( B \) are assumed to be Hermitian and \( B \) is also positive definite.

It uses a divide and conquer algorithm.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **itype**
  
  Must be 1 or 2 or 3. Specifies the problem type to be solved:
  
  - if \( \text{itype} = 1 \), the problem type is \( A\mathbf{x} = \lambda B\mathbf{x} \);
  - if \( \text{itype} = 2 \), the problem type is \( A^*B\mathbf{x} = \lambda\mathbf{x} \);
  - if \( \text{itype} = 3 \), the problem type is \( B^*A\mathbf{x} = \lambda\mathbf{x} \).

- **jobz**
  
  Must be 'N' or 'V'.
  
  - If \( \text{jobz} = 'N' \), then compute eigenvalues only.
  - If \( \text{jobz} = 'V' \), then compute eigenvalues and eigenvectors.

- **uplo**
  
  Must be 'U' or 'L'.
  
  - If \( \text{uplo} = 'U' \), arrays \( a \) and \( b \) store the upper triangles of \( A \) and \( B \);
  - If \( \text{uplo} = 'L' \), arrays \( a \) and \( b \) store the lower triangles of \( A \) and \( B \).

- **n**
  
  The order of the matrices \( A \) and \( B \) (n\geq 0).

- **a, b**
  
  Arrays:
  
  - \( a \) (size at least \( \max(1, lda*n) \)) contains the upper or lower triangle of the Hermitian matrix \( A \), as specified by \( \text{uplo} \).
  - \( b \) (size at least \( \max(1, ldb*n) \)) contains the upper or lower triangle of the Hermitian positive definite matrix \( B \), as specified by \( \text{uplo} \).

- **lda**
  
  The leading dimension of \( a \); at least \( \max(1, n) \).

- **ldb**
  
  The leading dimension of \( b \); at least \( \max(1, n) \).

**Output Parameters**

- **a**
  
  On exit, if \( \text{jobz} = 'V' \), then if \( \text{info} = 0 \), \( a \) contains the matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:
  
  - if \( \text{itype} = 1 \) or 2, \( Z^*BZ = I \);
if \( i_{\text{type}} = 3 \), \( Z^H \text{inv}(B) Z = I \);

If \( \text{jobz} = 'N' \), then on exit the upper triangle (if \( \text{uplo} = 'U' \)) or the lower triangle (if \( \text{uplo} = 'L' \)) of \( A \), including the diagonal, is destroyed.

On exit, if \( \text{info} \leq n \), the part of \( b \) containing the matrix is overwritten by the triangular factor \( U \) or \( L \) from the Cholesky factorization \( B = U^H U \) or \( B = L^H L \).

Array, size at least \( \max(1, n) \).

If \( \text{info} = 0 \), contains the eigenvalues in ascending order.

Return Values
This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), and \( \text{jobz} = 'N' \), then the algorithm failed to converge; \( i \)-off-diagonal elements of an intermediate tridiagonal form did not converge to zero;

if \( \text{info} = i \), and \( \text{jobz} = 'V' \), then the algorithm failed to compute an eigenvalue while working on the submatrix lying in rows and columns \( \text{info}/(n+1) \) through \( \text{mod}(\text{info}, n+1) \).

If \( \text{info} = n + i \), for \( 1 \leq i \leq n \), then the leading minor of order \( i \) of \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

sygvx
Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.

Syntax

```c
#include <mkl.h>

lapack_int LAPACKE_ssygvx (int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_dsygvx (int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);
```

Include Files

- mkl.h

Description
The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[ A^* x = \lambda^* B^* x, \quad A^* B^* x = \lambda^* x, \quad \text{or} \quad B^* A^* x = \lambda^* x. \]

Here \( A \) and \( B \) are assumed to be symmetric and \( B \) is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.
Input Parameters

**matrix_layout**
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**itype**
Must be 1 or 2 or 3. Specifies the problem type to be solved:
if itype = 1, the problem type is $A*x = \lambda*B*x$;
if itype = 2, the problem type is $A*B*x = \lambda*x$;
if itype = 3, the problem type is $B*A*x = \lambda*x$.

**jobz**
Must be 'N' or 'V'.
If jobz = 'N', then compute eigenvalues only.
If jobz = 'V', then compute eigenvalues and eigenvectors.

**range**
Must be 'A' or 'V' or 'I'.
If range = 'A', the routine computes all eigenvalues.
If range = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:
$vl < w[i] \leq vu$.
If range = 'I', the routine computes eigenvalues with indices $il$ to $iu$.

**uplo**
Must be 'U' or 'L'.
If uplo = 'U', arrays a and b store the upper triangles of $A$ and $B$;
If uplo = 'L', arrays a and b store the lower triangles of $A$ and $B$.

**n**
The order of the matrices $A$ and $B$ ($n \geq 0$).

**a, b**
Arrays:
a (size at least max(1, lda*n)) contains the upper or lower triangle of the
symmetric matrix $A$, as specified by uplo.
b (size at least max(1, ldb*n)) contains the upper or lower triangle of the
symmetric positive definite matrix $B$, as specified by uplo.

**lda**
The leading dimension of a; at least max(1, n).

**ldb**
The leading dimension of b; at least max(1, n).

**vl, vu**
If range = 'V', the lower and upper bounds of the interval to be searched
for eigenvalues.
Constraint: $vl < vu$.
If range = 'A' or 'I', vl and vu are not referenced.

**il, iu**
If range = 'I', the indices in ascending order of the smallest and largest
eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$
if $n = 0$.
If range = 'A' or 'V', il and iu are not referenced.
The leading dimension of the output array `z`. Constraints:
\[ ldz \geq 1; \text{if } jobz = 'V', ldz \geq \max(1, n) \text{ for column major layout and } ldz \geq \max(1, m) \text{ for row major layout.} \]

**Output Parameters**

- **a**: On exit, the upper triangle (if `uplo = 'U'`) or the lower triangle (if `uplo = 'L'`) of `A`, including the diagonal, is overwritten.

- **b**: On exit, if `info \leq n`, the part of `b` containing the matrix is overwritten by the triangular factor `U` or `L` from the Cholesky factorization `B = U^T * U` or `B = L * L^T`.

- **m**: The total number of eigenvalues found, \[ 0 \leq m \leq n. \text{ If } range = 'A', m = n, \text{ and if } range = 'I', m = iu - il + 1. \]

- **w, z**: Arrays:
  - `w`, size at least \( \max(1, n) \).
  - The first \( m \) elements of `w` contain the selected eigenvalues in ascending order.
  - `z` (size at least \( \max(1, ldz * m) \) for column major layout and \( \max(1, ldz * n) \) for row major layout).
  - If `jobz = 'V'`, then if `info = 0`, the first \( m \) columns of `z` contain the orthonormal eigenvectors of the matrix `A` corresponding to the selected eigenvalues, with the \( i \)-th column of `z` holding the eigenvector associated with \( w[i - 1] \). The eigenvectors are normalized as follows:
    - if `itype = 1` or `2`, \( Z^T * B * Z = I \);
    - if `itype = 3`, \( Z^T * \text{inv}(B) * Z = I \);
  - If `jobz = 'N'`, then `z` is not referenced.

- **ifail**: Array, size at least \( \max(1, n) \).
  - If `jobz = 'V'`, then if `info = 0`, the first \( m \) elements of `ifail` are zero; if `info > 0`, the `ifail` contains the indices of the eigenvectors that failed to converge.
  - If `jobz = 'N'`, then `ifail` is not referenced.

**Return Values**

This function returns a value `info`.
If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info > 0, spotrf/dpotrf and ssyevx/dsyevx returned an error code:

If info = i ≤ n, ssyevx/dsyevx failed to converge, and i eigenvectors failed to converge. Their indices are stored in the array ifail;

If info = n + i, for 1 ≤ i ≤ n, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes
An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width less than or equal to abstol+ε*max(|a|,|b|), where ε is the machine precision.

If abstol is less than or equal to zero, then ε*||T||_1 is used as tolerance, where T is the tridiagonal matrix obtained by reducing C to tridiagonal form, where C is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when abstol is set to twice the underflow threshold 2*?lamch('S'), not zero.

If this routine returns with info > 0, indicating that some eigenvectors did not converge, set abstol to 2*?lamch('S').

?hegvx
Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.

Syntax

lapack_int LAPACKE_chegvx( int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );

lapack_int LAPACKE_zhegvx( int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w, lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );

Include Files
• mkl.h

Description
The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

A*x = λ*B*x,  A*B*x = λ*x, or B*A*x = λ*x.

Here A and B are assumed to be Hermitian and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.
Input Parameters

**matrix_layout**

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

**itype**

Must be 1 or 2 or 3. Specifies the problem type to be solved:

- if itype = 1, the problem type is $A*x = \lambda*B*x$;
- if itype = 2, the problem type is $A*B*x = \lambda*x$;
- if itype = 3, the problem type is $B*A*x = \lambda*x$.

**jobz**

Must be 'N' or 'V'.

- If jobz = 'N', then compute eigenvalues only.
- If jobz = 'V', then compute eigenvalues and eigenvectors.

**range**

Must be 'A' or 'V' or 'I'.

- If range = 'A', the routine computes all eigenvalues.
- If range = 'V', the routine computes eigenvalues $w[i]$ in the half-open interval:
  $vl < w[i] \leq vu$.
- If range = 'I', the routine computes eigenvalues with indices $il$ to $iu$.

**uplo**

Must be 'U' or 'L'.

- If uplo = 'U', arrays a and b store the upper triangles of A and B;
- If uplo = 'L', arrays a and b store the lower triangles of A and B.

**n**

The order of the matrices $A$ and $B$ ($n \geq 0$).

**a**, **b**

Arrays:

- a (size at least max(1, lda*n)) contains the upper or lower triangle of the Hermitian matrix $A$, as specified by uplo.
- b (size at least max(1, ldb*n)) contains the upper or lower triangle of the Hermitian positive definite matrix $B$, as specified by uplo.

**lda**

The leading dimension of a; at least max(1, n).

**ldb**

The leading dimension of b; at least max(1, n).

**vl**, **vu**

If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: $vl < vu$.

If range = 'A' or 'I', vl and vu are not referenced.

**il**, **iu**

If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il=1$ and $iu=0$ if $n = 0$.

If range = 'A' or 'V', il and iu are not referenced.
**abstol**

The absolute error tolerance for the eigenvalues. See Application Notes for more information.

**ldz**

The leading dimension of the output array z. Constraints:

\[ ldz \geq 1; \text{if } jobz = 'V', ldz \geq \max(1, n) \text{ for column major layout and } ldz \geq \max(1, m) \text{ for row major layout.} \]

### Output Parameters

**a**

On exit, the upper triangle (if uplo = 'U') or the lower triangle (if uplo = 'L') of A, including the diagonal, is overwritten.

**b**

On exit, if info\(\leq n\), the part of b containing the matrix is overwritten by the triangular factor U or L from the Cholesky factorization \( B = U^H \) or \( B = L \). The total number of eigenvalues found,

\[ 0 \leq m \leq n. \text{ If range = 'A', } m = n, \text{ and if range = 'I', } m = iu-il+1. \]

**w**

Array, size at least \( \max(1, n) \).

The first \( m \) elements of w contain the selected eigenvalues in ascending order.

**z**

Array z(size at least \( \max(1, ldz*m) \) for column major layout and \( \max(1, ldz*n) \) for row major layout).

If jobz = 'V', then if info = 0, the first \( m \) columns of z contain the orthonormal eigenvectors of the matrix A corresponding to the selected eigenvalues, with the \( i \)-th column of z holding the eigenvector associated with \( w[i-1] \). The eigenvectors are normalized as follows:

- If itype = 1 or 2, \( Z^H*B*Z = I \);
- If itype = 3, \( Z^H*inv(B)*Z = I \);

If jobz = 'N', then z is not referenced.

If an eigenvector fails to converge, then that column of z contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

Note: you must ensure that at least \( \max(1,m) \) columns are supplied in the array z; if range = 'V', the exact value of m is not known in advance and an upper bound must be used.

**ifail**

Array, size at least \( \max(1, n) \).

If jobz = 'V', then if info = 0, the first \( m \) elements of ifail are zero; if info > 0, the ifail contains the indices of the eigenvectors that failed to converge.

If jobz = 'N', then ifail is not referenced.

### Return Values

This function returns a value \( \text{info} \).
If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} > 0 \), \text{cpotrf}/\text{zpotrf} \) and \( \text{cheevx}/\text{zheevx} \) returned an error code:

If \( \text{info} = i \leq n \), \text{cheevx}/\text{zheevx} failed to converge, and \( i \) eigenvectors failed to converge. Their indices are stored in the array \( \text{ifail} \);

If \( \text{info} = n + i \), for \( 1 \leq i \leq n \), then the leading minor of order \( i \) of \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

**Application Notes**

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a,b]\) of width less than or equal to \( \text{abstol}+\varepsilon \times \max(|a|,|b|) \), where \( \varepsilon \) is the machine precision.

If \( \text{abstol} \) is less than or equal to zero, then \( \varepsilon \times \|T\|_1 \) will be used in its place, where \( T \) is the tridiagonal matrix obtained by reducing \( C \) to tridiagonal form, where \( C \) is the symmetric matrix of the standard symmetric problem to which the generalized problem is transformed. Eigenvalues will be computed most accurately when \( \text{abstol} \) is set to twice the underflow threshold \( 2 \times \text{?lamch}('S') \), not zero.

If this routine returns with \( \text{info} > 0 \), indicating that some eigenvectors did not converge, try setting \( \text{abstol} \) to \( 2 \times \text{?lamch}('S') \).

**?spgv**

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

**Syntax**

```c
lapack_int LAPACK_sspgv (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, float* ap, float* bp, float* w, float* z, lapack_int ldz);
lapack_int LAPACK_dspgv (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, double* ap, double* bp, double* w, double* z, lapack_int ldz);
```

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[
A^* x = \lambda^* B^* x, \quad A^* B^* x = \lambda^* x, \quad \text{or} \quad B^* A^* x = \lambda^* x.
\]

Here \( A \) and \( B \) are assumed to be symmetric, stored in packed format, and \( B \) is also positive definite.

**Input Parameters**

- **matrix_layout**
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **itype**
  - Must be 1 or 2 or 3. Specifies the problem type to be solved:
    - if \( \text{itype} = 1 \), the problem type is \( A^* x = \lambda^* B^* x \);
    - if \( \text{itype} = 2 \), the problem type is \( A^* B^* x = \lambda^* x \);
if $i_{\text{type}} = 3$, the problem type is $B^*A^*x = \lambda x$.

jobz

Must be 'N' or 'V'.
If jobz = 'N', then compute eigenvalues only.
If jobz = 'V', then compute eigenvalues and eigenvectors.

uplo

Must be 'U' or 'L'.
If uplo = 'U', arrays $ap$ and $bp$ store the upper triangles of $A$ and $B$;
If uplo = 'L', arrays $ap$ and $bp$ store the lower triangles of $A$ and $B$.

$n$

The order of the matrices $A$ and $B$ ($n \geq 0$).

$ap, bp$

Arrays:
$ap$ contains the packed upper or lower triangle of the symmetric matrix $A$,
as specified by uplo.
The dimension of $ap$ must be at least max(1, $n * (n+1)/2$).
$bp$ contains the packed upper or lower triangle of the symmetric matrix $B$,
as specified by uplo.
The dimension of $bp$ must be at least max(1, $n * (n+1)/2$).

$ldz$

The leading dimension of the output array $z$; $ldz \geq 1$. If jobz = 'V', $ldz \geq$
max(1, $n$).

Output Parameters

$ap$

On exit, the contents of $ap$ are overwritten.

$bp$

On exit, contains the triangular factor $U$ or $L$ from the Cholesky factorization
$B = U^T U$ or $B = L L^T$, in the same storage format as $B$.

$w, z$

Arrays:
$w$, size at least max(1, $n$).
If info = 0, contains the eigenvalues in ascending order.
$z$ (size max(1, $ldz * n$)).
If jobz = 'V', then if info = 0, $z$ contains the matrix $Z$ of eigenvectors.
The eigenvectors are normalized as follows:
if $i_{\text{type}} = 1$ or $2$, $Z^T B^* Z = I$;
if $i_{\text{type}} = 3$, $Z^T \text{inv}(B) * Z = I$;
If jobz = 'N', then $z$ is not referenced.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the $i$-th parameter had an illegal value.
If info > 0, $\text{spptrf/dpptrf}$ and $\text{sspev/dspev}$ returned an error code:
If $info = i \leq n$, sspev/dspev failed to converge, and $i$ off-diagonal elements of an intermediate tridiagonal
did not converge to zero;

If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order $i$ of $B$ is not positive-definite. The factorization
of $B$ could not be completed and no eigenvalues or eigenvectors were computed.

?hpgv
Computes all eigenvalues and, optionally,
eigenvectors of a complex generalized Hermitian
positive-definite eigenproblem with matrices in packed
storage.

Syntax

lapack_int LAPACKE_chpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhpgv( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
lapack_complex_double* z, lapack_int ldz );

Include Files
• mkl.h

Description
The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized
Hermitian positive-definite eigenproblem, of the form

$A^*x = \lambda B^*x$, $A^*B^*x = \lambda x$, or $B^*A^*x = \lambda x$.

Here $A$ and $B$ are assumed to be Hermitian, stored in packed format, and $B$ is also positive definite.

Input Parameters

$matrix_layout$ Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR)
or column major (LAPACK_COL_MAJOR).

$itype$ Must be 1 or 2 or 3. Specifies the problem type to be solved:
if $itype = 1$, the problem type is $A^*x = \lambda B^*x$;
if $itype = 2$, the problem type is $A^*B^*x = \lambda x$;
if $itype = 3$, the problem type is $B^*A^*x = \lambda x$.

$jobz$ Must be 'N' or 'V'.
If $jobz = 'N'$, then compute eigenvalues only.
If $jobz = 'V'$, then compute eigenvalues and eigenvectors.

$uplo$ Must be 'U' or 'L'.
If $uplo = 'U'$, arrays $ap$ and $bp$ store the upper triangles of $A$ and $B$;
If $uplo = 'L'$, arrays $ap$ and $bp$ store the lower triangles of $A$ and $B$.

$n$ The order of the matrices $A$ and $B$ ($n \geq 0$).
Arrays:
ap contains the packed upper or lower triangle of the Hermitian matrix \( A \), as specified by \( uplo \).

The dimension of \( ap \) must be at least \( \max(1, n*(n+1)/2) \).

\( bp \) contains the packed upper or lower triangle of the Hermitian matrix \( B \), as specified by \( uplo \).

The dimension of \( bp \) must be at least \( \max(1, n*(n+1)/2) \).

\( ldz \):

The leading dimension of the output array \( z \); \( ldz \geq 1 \). If \( jobz = 'V' \), \( ldz \geq \max(1, n) \).

Output Parameters

\( ap \):

On exit, the contents of \( ap \) are overwritten.

\( bp \):

On exit, contains the triangular factor \( U \) or \( L \) from the Cholesky factorization \( B = U^H \cdot U \) or \( B = L \cdot L^H \), in the same storage format as \( B \).

\( w \):

Array, size at least \( \max(1, n) \).

If \( info = 0 \), contains the eigenvalues in ascending order.

\( z \):

Array \( z \) (size \( \max(1, ldz * n) \)).

If \( jobz = 'V' \), then if \( info = 0 \), \( z \) contains the matrix \( Z \) of eigenvectors.
The eigenvectors are normalized as follows:

- If \( itype = 1 \) or \( 2 \), \( Z^H \cdot B \cdot Z = I \);
- If \( itype = 3 \), \( Z^H \cdot inv(B) \cdot Z = I \);
- If \( jobz = 'N' \), then \( z \) is not referenced.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info > 0 \), \( cpptrf/zpttrf \) and \( chpev/zhpev \) returned an error code:

If \( info = i \leq n \), \( chpev/zhpev \) failed to converge, and \( i \) off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If \( info = n + i \), for \( 1 \leq i \leq n \), then the leading minor of order \( i \) of \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

\( ? \)spgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

```c
lapack_int LAPACKE_sspgvd (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, float* ap, float* bp, float* w, float* z, lapack_int ldz);
```
lapack_int LAPACKE_dspgvd (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, double* ap, double* bp, double* w, double* z, lapack_int ldz);

**Include Files**
- mkl.h

**Description**

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[ A^* x = \lambda^* B^* x, \quad A^* B^* x = \lambda^* x, \quad \text{or} \quad B^* A^* x = \lambda^* x. \]

Here \( A \) and \( B \) are assumed to be symmetric, stored in packed format, and \( B \) is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **itype**
  Must be 1 or 2 or 3. Specifies the problem type to be solved:
  - if `itype = 1`, the problem type is \( A^* x = \lambda^* B^* x \);
  - if `itype = 2`, the problem type is \( A^* B^* x = \lambda^* x \);
  - if `itype = 3`, the problem type is \( B^* A^* x = \lambda^* x \).

- **jobz**
  Must be `'N'` or `'V'`.
  - If `jobz = 'N'`, then compute eigenvalues only.
  - If `jobz = 'V'`, then compute eigenvalues and eigenvectors.

- **uplo**
  Must be `'U'` or `'L'`.
  - If `uplo = 'U'`, arrays `ap` and `bp` store the upper triangles of \( A \) and \( B \);
  - If `uplo = 'L'`, arrays `ap` and `bp` store the lower triangles of \( A \) and \( B \).

- **n**
  The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

- **ap, bp**
  Arrays:
  - `ap` contains the packed upper or lower triangle of the symmetric matrix \( A \), as specified by `uplo`.
  - `bp` contains the packed upper or lower triangle of the symmetric matrix \( B \), as specified by `uplo`.

  The dimension of `ap` must be at least \( \max(1, n(n+1)/2) \).
  The dimension of `bp` must be at least \( \max(1, n(n+1)/2) \).

- **ldz**
  The leading dimension of the output array `z`; `ldz` \( \geq 1 \). If `jobz = 'V'`, `ldz` \( \geq \max(1, n) \).

**Output Parameters**

- **ap**
  On exit, the contents of `ap` are overwritten.
On exit, contains the triangular factor $U$ or $L$ from the Cholesky factorization
$B = U^T U$ or $B = L^T L$, in the same storage format as $B$.

Arrays:

$w$, size at least max(1, $n$).
If $info = 0$, contains the eigenvalues in ascending order.
$z$ (size at least max(1, $ldz*n$)).
If $jobz = 'V$', then if $info = 0$, $z$ contains the matrix $Z$ of eigenvectors.
The eigenvectors are normalized as follows:
if $itype = 1$ or $2$, $Z^T B^* Z = I$;
if $itype = 3$, $Z^T \text{inv}(B)^* Z = I$;
If $jobz = 'N$', then $z$ is not referenced.

Return Values

This function returns a value $info$.
If $info=0$, the execution is successful.
If $info = -i$, the $i$-th parameter had an illegal value.
If $info > 0$, spptrf/dptrf and sspevd/dspevd returned an error code:
If $info = i \leq n$, sspevd/dspevd failed to converge, and $i$ off-diagonal elements of an intermediate tridiagonal
did not converge to zero;
If $info = n + i$, for $1 \leq i \leq n$, then the leading minor of order $i$ of $B$ is not positive-definite. The factorization
of $B$ could not be completed and no eigenvalues or eigenvectors were computed.

?hpgvd
Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with matrices in packed storage using a divide and conquer method.

Syntax

lapack_int LAPACKE_chpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float* w,
lapack_complex_float* z, lapack_int ldz);

lapack_int LAPACKE_zhpgvd( int matrix_layout, lapack_int itype, char jobz, char uplo,
lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double* w,
lapack_complex_double* z, lapack_int ldz);

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

$A^* x = \lambda^* B^* x$, $A^* B^* x = \lambda^* x$, or $B^* A^* x = \lambda^* x$.

Here $A$ and $B$ are assumed to be Hermitian, stored in packed format, and $B$ is also positive definite.
If eigenvectors are desired, it uses a divide and conquer algorithm.

**Input Parameters**

**matrix_layout**

Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

**itype**

Must be 1 or 2 or 3. Specifies the problem type to be solved:

- If itype = 1, the problem type is $A*x = \lambda*B*x$;
- If itype = 2, the problem type is $A*B*x = \lambda*x$;
- If itype = 3, the problem type is $B*A*x = \lambda*B*x$.

**jobz**

Must be 'N' or 'V'.

- If jobz = 'N', then compute eigenvalues only.
- If jobz = 'V', then compute eigenvalues and eigenvectors.

**uplo**

Must be 'U' or 'L'.

- If uplo = 'U', arrays $ap$ and $bp$ store the upper triangles of $A$ and $B$;
- If uplo = 'L', arrays $ap$ and $bp$ store the lower triangles of $A$ and $B$.

**n**

The order of the matrices $A$ and $B$ ($n \geq 0$).

**ap, bp**

Arrays:

- $ap$ contains the packed upper or lower triangle of the Hermitian matrix $A$, as specified by uplo.
  - The dimension of $ap$ must be at least max(1, $n*(n+1)/2$).
- $bp$ contains the packed upper or lower triangle of the Hermitian matrix $B$, as specified by uplo.
  - The dimension of $bp$ must be at least max(1, $n*(n+1)/2$).

**ldz**

The leading dimension of the output array $z$; $ldz \geq 1$. If jobz = 'V', $ldz \geq \max(1, n)$.

**Output Parameters**

**ap**

On exit, the contents of $ap$ are overwritten.

**bp**

On exit, contains the triangular factor $U$ or $L$ from the Cholesky factorization $B = U^H*U$ or $B = L*L^H$, in the same storage format as $B$.

**w**

Array, size at least max(1, $n$).

- If info = 0, contains the eigenvalues in ascending order.

**z**

Array $z$ (size at least max(1, $ldz*n$)).

- If jobz = 'V', then if info = 0, $z$ contains the matrix $Z$ of eigenvectors. The eigenvectors are normalized as follows:
  - if itype = 1 or 2, $Z^H*B*Z = I$;
  - if itype = 3, $Z^H*inv(B)*Z = I$;
- If jobz = 'N', then $z$ is not referenced.
Return Values

This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info > 0, cppyf/zpptrf and chpevd/zhpevd returned an error code:

If info = i≤n, chpevd/zhpevd failed to converge, and i off-diagonal elements of an intermediate tridiagonal did not converge to zero;

If info = n + i, for 1 ≤i≤n, then the leading minor of order i of B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?spgvx

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with matrices in packed storage.

Syntax

lapack_int LAPACKE_sspgvx (int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, float* ap, float* bp, float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_dspgvx (int matrix_layout, lapack_int itype, char jobz, char range, char uplo, lapack_int n, double* ap, double* bp, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, lapack_int* ifail);

Include Files

- mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[ A*x = \lambda*B*x, \quad A*B*x = \lambda*x, \quad \text{or} \quad B*A*x = \lambda*x. \]

Here A and B are assumed to be symmetric, stored in packed format, and B is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

matrix_layout

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

itype

Must be 1 or 2 or 3. Specifies the problem type to be solved:

if itype = 1, the problem type is \( A*x = \lambda*B*x \);

if itype = 2, the problem type is \( A*B*x = \lambda*x \);

if itype = 3, the problem type is \( B*A*x = \lambda*x \).

jobz

Must be 'N' or 'V'.


If $jobz = 'N'$, then compute eigenvalues only.
If $jobz = 'V'$, then compute eigenvalues and eigenvectors.

$range$  
Must be 'A' or 'V' or 'I'.
If $range = 'A'$, the routine computes all eigenvalues.
If $range = 'V'$, the routine computes eigenvalues $w[i]$ in the half-open interval:
$vl < w[i] \leq vu$.
If $range = 'I'$, the routine computes eigenvalues with indices $il$ to $iu$.

$uplo$  
Must be 'U' or 'L'.
If $uplo = 'U'$, arrays $ap$ and $bp$ store the upper triangles of $A$ and $B$;
If $uplo = 'L'$, arrays $ap$ and $bp$ store the lower triangles of $A$ and $B$.

$n$  
The order of the matrices $A$ and $B$ ($n \geq 0$).

$ap$, $bp$  
Arrays:
$ap$ contains the packed upper or lower triangle of the symmetric matrix $A$, as specified by $uplo$.
The size of $ap$ must be at least $\max(1, n^*(n+1)/2)$.
$bp$ contains the packed upper or lower triangle of the symmetric matrix $B$, as specified by $uplo$.
The size of $bp$ must be at least $\max(1, n^*(n+1)/2)$.

$vl$, $vu$  
If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.
Constraint: $vl < vu$.
If $range = 'A'$ or 'I', $vl$ and $vu$ are not referenced.

$il$, $iu$  
If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned.
Constraint: $1 \leq il \leq iu \leq n$, if $n > 0$; $il = 1$ and $iu = 0$ if $n = 0$.
If $range = 'A'$ or 'V', $il$ and $iu$ are not referenced.

$abstol$  
The absolute error tolerance for the eigenvalues. See Application Notes for more information.

$ldz$  
The leading dimension of the output array $z$. Constraints:
$ldz \geq 1$; if $jobz = 'V'$, $ldz \geq \max(1, n)$ for column major layout and $ldz \geq \max(1, m)$ for row major layout.

Output Parameters

$ap$  
On exit, the contents of $ap$ are overwritten.
On exit, contains the triangular factor $U$ or $L$ from the Cholesky factorization $B = U^T U$ or $B = L^T L^T$, in the same storage format as $B$.

The total number of eigenvalues found, $0 \leq m \leq n$. If range = 'A', $m = n$, and if range = 'I', $m = iu-il+1$.

Arrays:
$w$, size at least max(1, $n$).
If info = 0, contains the eigenvalues in ascending order.
$z$ (size at least max(1, ldz* $m$) for column major layout and max(1, ldz* $n$) for row major layout).
If jobz = 'V', then if info = 0, the first $m$ columns of $z$ contain the orthonormal eigenvectors of the matrix $A$ corresponding to the selected eigenvalues, with the $i$-th column of $z$ holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:
if itype = 1 or 2, $Z^T B Z = I$;
if itype = 3, $Z^T \text{inv}(B) Z = I$;
If jobz = 'N', then $z$ is not referenced.
If an eigenvector fails to converge, then that column of $z$ contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

Note: you must ensure that at least max(1, $m$) columns are supplied in the array $z$; if range = 'V', the exact value of $m$ is not known in advance and an upper bound must be used.

$\text{ifail}$
Array, size at least max(1, $n$).
If jobz = 'V', then if info = 0, the first $m$ elements of $\text{ifail}$ are zero; if info > 0, the $\text{ifail}$ contains the indices of the eigenvectors that failed to converge.
If jobz = 'N', then $\text{ifail}$ is not referenced.

**Return Values**
This function returns a value info.
If info=0, the execution is successful.
If info = -i, the $i$-th parameter had an illegal value.
If info > 0, spptrf/dpptrf and sspevx/dspevx returned an error code:
If info = $i$≤$n$, sspevx/dspevx failed to converge, and $i$ eigenvectors failed to converge. Their indices are stored in the array ifail;
If info = $n + i$, for 1 ≤$i$≤$n$, then the leading minor of order $i$ of $B$ is not positive-definite. The factorization of $B$ could not be completed and no eigenvalues or eigenvectors were computed.
Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a,b]\) of width less than or equal to \(\text{abstol} + \varepsilon \max(|a|,|b|)\), where \(\varepsilon\) is the machine precision.

If \(\text{abstol}\) is less than or equal to zero, then \(\varepsilon \|T\|_1\) is used instead, where \(T\) is the tridiagonal matrix obtained by reducing \(A\) to tridiagonal form. Eigenvalues are computed most accurately when \(\text{abstol}\) is set to twice the underflow threshold \(2^{*}\text{lamch}'S'\), not zero.

If this routine returns with \(\text{info} > 0\), indicating that some eigenvectors did not converge, set \(\text{abstol}\) to \(2^{*}\text{lamch}'S'\).

\(\text{chpgvx}\)

Computes selected eigenvalues and, optionally, eigenvectors of a generalized Hermitian positive-definite eigenproblem with matrices in packed storage.

Syntax

```
lapack_int LAPACKE_chpgvx( int matrix_layout, lapack_int itype, char jobz, char range,    char uplo, lapack_int n, lapack_complex_float* ap, lapack_complex_float* bp, float vl,    float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,    lapack_complex_float* z, lapack_int ldz, lapack_int* ifail );
```

```
lapack_int LAPACKE_zhpgvx( int matrix_layout, lapack_int itype, char jobz, char range,    char uplo, lapack_int n, lapack_complex_double* ap, lapack_complex_double* bp, double vl,    double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,    lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- \(\text{mkl.h}\)

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

\[ A^*x = \lambda^*B^*x, \quad A^*B^*x = \lambda^*x, \quad \text{or} \quad B^*A^*x = \lambda^*x. \]

Here \(A\) and \(B\) are assumed to be Hermitian, stored in packed format, and \(B\) is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **itype**
  Must be 1 or 2 or 3. Specifies the problem type to be solved:
  - if \(\text{itype} = 1\), the problem type is \(A^*x = \lambda^*B^*x\);
  - if \(\text{itype} = 2\), the problem type is \(A^*B^*x = \lambda^*x\);
  - if \(\text{itype} = 3\), the problem type is \(B^*A^*x = \lambda^*x\).

- **jobz**
  Must be ‘N’ or ‘V’.
  - If \(\text{jobz} = 'N'\), then compute eigenvalues only.
If \( \text{jobz} = 'V' \), then compute eigenvalues and eigenvectors.

**range**

Must be 'A' or 'V' or 'I'.

If \( \text{range} = 'A' \), the routine computes all eigenvalues.

If \( \text{range} = 'V' \), the routine computes eigenvalues \( w[i] \) in the half-open interval:

\[ vl < w[i] \leq vu. \]

If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).

**uplo**

Must be 'U' or 'L'.

If \( \text{uplo} = 'U' \), arrays \( ap \) and \( bp \) store the upper triangles of \( A \) and \( B \);

If \( \text{uplo} = 'L' \), arrays \( ap \) and \( bp \) store the lower triangles of \( A \) and \( B \).

**n**

The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

**ap, bp**

Arrays:

\( ap \) contains the packed upper or lower triangle of the Hermitian matrix \( A \), as specified by \( \text{uplo} \).

The dimension of \( ap \) must be at least \( \max(1, n*(n+1)/2) \).

\( bp \) contains the packed upper or lower triangle of the Hermitian matrix \( B \), as specified by \( \text{uplo} \).

The dimension of \( bp \) must be at least \( \max(1, n*(n+1)/2) \).

**vl, vu**

If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If \( \text{range} = 'A' \) or 'I', \( vl \) and \( vu \) are not referenced.

**il, iu**

If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il=1 \) and \( iu=0 \) if \( n = 0 \).

If \( \text{range} = 'A' \) or 'V', \( il \) and \( iu \) are not referenced.

**abstol**

The absolute error tolerance for the eigenvalues.

See Application Notes for more information.

**ldz**

The leading dimension of the output array \( z \); \( ldz \geq 1 \). If \( \text{jobz} = 'V' \), \( ldz \geq \max(1, n) \) for column major layout and \( ldz \geq \max(1, m) \) for row major layout.

### Output Parameters

**ap**

On exit, the contents of \( ap \) are overwritten.

**bp**

On exit, contains the triangular factor \( U \) or \( L \) from the Cholesky factorization \( B = U^H U \) or \( B = L L^H \), in the same storage format as \( B \).
The total number of eigenvalues found, $0 \leq m \leq n$. If \textit{range} = 'A', $m = n$, and if \textit{range} = 'I', $m = \text{iu} - \text{il} + 1$.

$w$

Array, size at least $\max(1, n)$.
If \textit{info} = 0, contains the eigenvalues in ascending order.

$z$

Array $z$(size at least $\max(1, \text{ldz}*m)$ for column major layout and $\max(1, \text{ldz}*n)$ for row major layout).
If \textit{jobz} = 'V', then if \textit{info} = 0, the first $m$ columns of $z$ contain the orthonormal eigenvectors of the matrix $A$ corresponding to the selected eigenvalues, with the $i$-th column of $z$ holding the eigenvector associated with $w(i)$. The eigenvectors are normalized as follows:

- If $\text{itype} = 1$ or $2$, $Z^H B^H Z = I$;
- If $\text{itype} = 3$, $Z^H \text{inv}(B) Z = I$;
If \textit{jobz} = 'N', then $z$ is not referenced.

If an eigenvector fails to converge, then that column of $z$ contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \textit{ifail}.

Note: you must ensure that at least $\max(1, m)$ columns are supplied in the array $z$; if \textit{range} = 'V', the exact value of $m$ is not known in advance and an upper bound must be used.

$\textit{ifail}$

Array, size at least $\max(1, n)$.
If \textit{jobz} = 'V', then if \textit{info} = 0, the first $m$ elements of \textit{ifail} are zero; if \textit{info} > 0, the \textit{ifail} contains the indices of the eigenvectors that failed to converge.
If \textit{jobz} = 'N', then \textit{ifail} is not referenced.

Return Values
This function returns a value \textit{info}.
If \textit{info} = 0, the execution is successful.
If \textit{info} = -i, the $i$-th parameter had an illegal value.
If \textit{info} > 0, \texttt{cpptrf/zpptrf} and \texttt{chpevx/zhpevx} returned an error code:
If \textit{info} = $i \leq n$, \texttt{chpevx/zhpevx} failed to converge, and $i$ eigenvectors failed to converge. Their indices are stored in the array \textit{ifail};
If \textit{info} = $n + i$, for $1 \leq i \leq n$, then the leading minor of order $i$ of $B$ is not positive-definite. The factorization of $B$ could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes
An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a,b]$ of width less than or equal to $\text{abstol} + \varepsilon \max(|a|,|b|)$, where $\varepsilon$ is the machine precision.
If $\text{abstol}$ is less than or equal to zero, then $\varepsilon \|T\|_1$ is used as tolerance, where $T$ is the tridiagonal matrix obtained by reducing $A$ to tridiagonal form. Eigenvalues will be computed most accurately when $\text{abstol}$ is set to twice the underflow threshold $2*\text{?lamch('S')}$, not zero.
If this routine returns with \( \text{info} > 0 \), indicating that some eigenvectors did not converge, try setting \( \text{abstol} \) to \( 2^*?\text{lamch}(\text{'S'}) \).

\( \text{?sbgv} \)

_Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices._

**Syntax**

```c
lapack_int LAPACKE_ssbgv (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb, lapack_int ldbb, float* w, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dsbgv (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb, lapack_int ldbb, double* w, double* z, lapack_int ldz);
```

**Include Files**

- mkl.h

**Description**

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form \( A^*x = \lambda B^*x \). Here \( A \) and \( B \) are assumed to be symmetric and banded, and \( B \) is also positive definite.

**Input Parameters**

- `matrix_layout`
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- `jobz`
  Must be 'N' or 'V'.
  
  If `jobz` = 'N', then compute eigenvalues only.
  
  If `jobz` = 'V', then compute eigenvalues and eigenvectors.

- `uplo`
  Must be 'U' or 'L'.

  If `uplo` = 'U', arrays `ab` and `bb` store the upper triangles of \( A \) and \( B \);

  If `uplo` = 'L', arrays `ab` and `bb` store the lower triangles of \( A \) and \( B \).

- `n`
  The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

- `ka`
  The number of super- or sub-diagonals in \( A \) (\( ka \geq 0 \)).

- `kb`
  The number of super- or sub-diagonals in \( B \) (\( kb \geq 0 \)).

- `ab, bb`
  Arrays:

  \( ab \) (size at least max(1, \( ldab^*n \)) for column major layout and max(1, \( ldab^*(ka + 1) \)) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix \( A \) (as specified by `uplo`) in band storage format.
$bb$ (size at least max(1, $ldbb*\text{n}$) for column major layout and max(1, $ldbb*(kb + 1)$) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix $B$ (as specified by $\text{uplo}$) in band storage format.

$ldab$

The leading dimension of the array $ab$; must be at least $ka+1$ for column major layout and at least max(1, $n$) for row major layout.

$ldbb$

The leading dimension of the array $bb$; must be at least $kb+1$ for column major layout and at least max(1, $n$) for row major layout.

$ldz$

The leading dimension of the output array $z$; $ldz \geq 1$. If $jobz = 'V'$, $ldz \geq \text{max}(1, n)$.

**Output Parameters**

$ab$

On exit, the contents of $ab$ are overwritten.

$bb$

On exit, contains the factor $S$ from the split Cholesky factorization $B = S^T*S$, as returned by $\text{pbstf/pbstf}$.

$w$, $z$

Arrays:

$w$, size at least max(1, $n$).

If $info = 0$, contains the eigenvalues in ascending order.

$z$ (size at least max(1, $ldz*n$)).

If $jobz = 'V'$, then if $info = 0$, $z$ contains the matrix $Z$ of eigenvectors, with the $i$-th column of $z$ holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T*B*Z = I$.

If $jobz = 'N'$, then $z$ is not referenced.

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info > 0$, and

if $i \leq n$, the algorithm failed to converge, and $i$ off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if $info = n + i$, for $1 \leq i \leq n$, then $\text{pbstf/pbstf}$ returned $info = i$ and $B$ is not positive-definite. The factorization of $B$ could not be completed and no eigenvalues or eigenvectors were computed.

**?hbgv**

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.
Syntax

```c
lapack_int LAPACKE_chbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
  lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
  lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z,
  lapack_int ldz );

lapack_int LAPACKE_zhbgv( int matrix_layout, char jobz, char uplo, lapack_int n,
  lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
  lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z,
  lapack_int ldz );
```

Include Files

- mkl.h

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form $A^*x = \lambda^*B^*x$. Here $A$ and $B$ are Hermitian and banded matrices, and matrix $B$ is also positive definite.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **jobz**: Must be 'N' or 'V'.
  - If `jobz` = 'N', then compute eigenvalues only.
  - If `jobz` = 'V', then compute eigenvalues and eigenvectors.
- **uplo**: Must be 'U' or 'L'.
  - If `uplo` = 'U', arrays `ab` and `bb` store the upper triangles of $A$ and $B$.
  - If `uplo` = 'L', arrays `ab` and `bb` store the lower triangles of $A$ and $B$.
- **n**: The order of the matrices $A$ and $B$ ($n \geq 0$).
- **ka**: The number of super- or sub-diagonals in $A$ ($ka \geq 0$).
- **kb**: The number of super- or sub-diagonals in $B$ ($kb \geq 0$).
- **ab, bb**: Arrays:
  - `ab` (size at least max(1, ldab*n) for column major layout and max(1, ldab*(ka + 1)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix $A$ (as specified by `uplo`) in band storage format.
  - `bb` (size at least max(1, ldbb*n) for column major layout and max(1, ldbb*(kb + 1)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix $B$ (as specified by `uplo`) in band storage format.
- **ldab**: The leading dimension of the array `ab`; must be at least `ka+1` for column major layout and at least max(1, `n` for row major layout.)
ldbb

The leading dimension of the array \( bb \); must be at least \( kb+1 \) for column major layout and at least \( \max(1, n) \) for row major layout.

ldz

The leading dimension of the output array \( z \); \( ldz \geq 1 \). If \( jobz = 'V' \), \( ldz \geq \max(1, n) \).

Output Parameters

\( ab \)

On exit, the contents of \( ab \) are overwritten.

\( bb \)

On exit, contains the factor \( S \) from the split Cholesky factorization \( B = S^H S \), as returned by \( \text{pbstf/pbstf} \).

\( w \)

Array, size at least \( \max(1, n) \).

If \( info = 0 \), contains the eigenvalues in ascending order.

\( z \)

Array \( z \) (size at least \( \max(1, ldz*n) \)).

If \( jobz = 'V' \), then if \( info = 0 \), \( z \) contains the matrix \( Z \) of eigenvectors, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w(i) \). The eigenvectors are normalized so that \( Z^H B Z = I \).

If \( jobz = 'N' \), then \( z \) is not referenced.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info > 0 \), and

if \( i \leq n \), the algorithm failed to converge, and \( i \) off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if \( info = n + i \), for \( 1 \leq i \leq n \), then \( \text{pbstf/pbstf} \) returned \( info = i \) and \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvd

Computes all eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices. If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```c
lapack_int LAPACKE_ssbgvd (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb, lapack_int ldbb, float* w, float* z, lapack_int ldz);
```

```c
lapack_int LAPACKE_dsbgvd (int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb, lapack_int ldbb, double* w, double* z, lapack_int ldz);
```

Include Files

- mkl.h
Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form $A^*x = \lambda B^*x$. Here $A$ and $B$ are assumed to be symmetric and banded, and $B$ is also positive definite. If eigenvectors are desired, it uses a divide and conquer algorithm.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **jobz**: Must be 'N' or 'V'.
  - If jobz = 'N', then compute eigenvalues only.
  - If jobz = 'V', then compute eigenvalues and eigenvectors.
- **uplo**: Must be 'U' or 'L'.
  - If uplo = 'U', arrays ab and bb store the upper triangles of $A$ and $B$;
  - If uplo = 'L', arrays ab and bb store the lower triangles of $A$ and $B$.
- **n**: The order of the matrices $A$ and $B$ ($n \geq 0$). The number of super- or sub-diagonals in $A$ ($ka \geq 0$).
- **ka**: The number of super- or sub-diagonals in $A$ ($ka \geq 0$).
- **kb**: The number of super- or sub-diagonals in $B$ ($kb \geq 0$).
- **ab, bb**: Arrays:
  - $ab$(size at least max(1, $ldab*n$) for column major layout and max(1, $ldab*(ka + 1)$) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix $A$ (as specified by uplo) in band storage format.
  - $bb$(size at least max(1, $ldbb*n$) for column major layout and max(1, $ldbb*(kb + 1)$) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix $B$ (as specified by uplo) in band storage format.
- **ldab**: The leading dimension of the array $ab$; must be at least $ka+1$ for column major layout and at least max(1, $n$) for row major layout.
- **ldbb**: The leading dimension of the array $bb$; must be at least $kb+1$ for column major layout and at least max(1, $n$) for row major layout.
- **ldz**: The leading dimension of the output array $z$; $ldz \geq 1$. If jobz = 'V', $ldz \geq$ max(1, $n$).

Output Parameters

- **ab**: On exit, the contents of $ab$ are overwritten.
- **bb**: On exit, contains the factor $S$ from the split Cholesky factorization $B = S^T S$, as returned by pbstf/pbstf.
Arrays:

\( w, z \)

\( w \), size at least \( \max(1, n) \).

If \( info = 0 \), contains the eigenvalues in ascending order.

\( z \) (size at least \( \max(1, ldz*n) \)).

If \( jobz = 'V' \), then if \( info = 0 \), \( z \) contains the matrix \( Z \) of eigenvectors, with the \( i \)-th column of \( z \) holding the eigenvector associated with \( w[i-1] \). The eigenvectors are normalized so that \( Z^T B^* Z = I \).

If \( jobz = 'N' \), then \( z \) is not referenced.

Return Values

This function returns a value \( info \).

If \( info=0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info > 0 \), and

if \( i \leq n \), the algorithm failed to converge, and \( i \) off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if \( info = n + i \), for \( 1 \leq i \leq n \), then \( \text{pbstf} / \text{pbstf} \) returned \( info = i \) and \( B \) is not positive-definite. The factorization of \( B \) could not be completed and no eigenvalues or eigenvectors were computed.

?hbgvd

Computes all eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

If eigenvectors are desired, it uses a divide and conquer method.

Syntax

```c
lapack_int LAPACKE_chbgvd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab, lapack_complex_float* bb, lapack_int ldbb, float* w, lapack_complex_float* z, lapack_int ldz );

lapack_int LAPACKE_zhbgvd( int matrix_layout, char jobz, char uplo, lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab, lapack_complex_double* bb, lapack_int ldbb, double* w, lapack_complex_double* z, lapack_int ldz );
```

Include Files

- \texttt{mkl.h}

Description

The routine computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form \( A^* x = \lambda^* B^* x \). Here \( A \) and \( B \) are assumed to be Hermitian and banded, and \( B \) is also positive definite.

If eigenvectors are desired, it uses a divide and conquer algorithm.
**Input Parameters**

*matrix_layout*  
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

*jobz*  
Must be 'N' or 'V'.
- If *jobz* = 'N', then compute eigenvalues only.
- If *jobz* = 'V', then compute eigenvalues and eigenvectors.

*uplo*  
Must be 'U' or 'L'.
- If *uplo* = 'U', arrays *ab* and *bb* store the upper triangles of *A* and *B*.
- If *uplo* = 'L', arrays *ab* and *bb* store the lower triangles of *A* and *B*.

*n*  
The order of the matrices *A* and *B* (*n* ≥ 0).

*ka*  
The number of super- or sub-diagonals in *A* (*ka* ≥ 0).

*kb*  
The number of super- or sub-diagonals in *B* (*kb* ≥ 0).

*ab*, *bb*  
Arrays:
- *ab* (size at least max(1, *ldab* *n*) for column major layout and max(1, *ldab*(*ka* + 1)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix *A* (as specified by *uplo*) in band storage format.
- *bb* (size at least max(1, *ldbb* *n*) for column major layout and max(1, *ldbb*(*kb* + 1)) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix *B* (as specified by *uplo*) in band storage format.

*ldab*  
The leading dimension of the array *ab*; must be at least *ka*+1.

*ldbb*  
The leading dimension of the array *bb*; must be at least *kb*+1.

*ldz*  
The leading dimension of the output array *z*; *ldz* ≥ 1. If *jobz* = 'V', *ldz* ≥ max(1, *n*).

**Output Parameters**

*ab*  
On exit, the contents of *ab* are overwritten.

*bb*  
On exit, contains the factor *S* from the split Cholesky factorization *B* = *S*H*S*, as returned by *pbstf/pbstf*.

*w*  
Array, size at least max(1, *n*).
- If *info* = 0, contains the eigenvalues in ascending order.
- If *jobz* = 'V', then if *info* = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*(*i*). The eigenvectors are normalized so that *Z*H*B*Z = I.
- If *jobz* = 'N', then *z* is not referenced.
Return Values
This function returns a value info.

If info=0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info > 0, and

if info = n + i, for 1 ≤ i ≤ n, then pbstf/pbstf returned info = i and B is not positive-definite. The factorization of B could not be completed and no eigenvalues or eigenvectors were computed.

?sbgvx
Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem with banded matrices.

Syntax
lapack_int LAPACKE_ssbgvx (int matrix_layout, char jobz, char range, char uplo, 
lapack_int n, lapack_int ka, lapack_int kb, float* ab, lapack_int ldab, float* bb, 
lapack_int ldbb, float* q, lapack_int ldq, float vl, float vu, lapack_int il, lapack_int iu, 
float abstol, lapack_int* m, float* w, float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_dsbgvx (int matrix_layout, char jobz, char range, char uplo, 
lapack_int n, lapack_int ka, lapack_int kb, double* ab, lapack_int ldab, double* bb, 
lapack_int ldbb, double* q, lapack_int ldq, double vl, double vu, lapack_int il, 
lapack_int iu, double abstol, lapack_int* m, double* w, double* z, lapack_int ldz, 
lapack_int* ifail);

Include Files
• mkl.h

Description
The routine computes selected eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite banded eigenproblem, of the form \( A x = \lambda B x \). Here \( A \) and \( B \) are assumed to be symmetric and banded, and \( B \) is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

matrix_layout
Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

jobz
Must be 'N' or 'V'.

If jobz = 'N', then compute eigenvalues only.

If jobz = 'V', then compute eigenvalues and eigenvectors.

range
Must be 'A' or 'V' or 'I'.

If range = 'A', the routine computes all eigenvalues.
If range = 'V', the routine computes eigenvalues \( w[i] \) in the half-open interval: 
\[ vl < w[i] \leq vu. \]

If range = 'I', the routine computes eigenvalues in range \( il \) to \( iu \).

\( uplo \) Must be 'U' or 'L'.

If \( uplo = 'U' \), arrays \( ab \) and \( bb \) store the upper triangles of \( A \) and \( B \);

If \( uplo = 'L' \), arrays \( ab \) and \( bb \) store the lower triangles of \( A \) and \( B \).

\( n \) The order of the matrices \( A \) and \( B \) \((n \geq 0)\).

\( ka \) The number of super- or sub-diagonals in \( A \)
\((ka \geq 0)\).

\( kb \) The number of super- or sub-diagonals in \( B \) \((kb \geq 0)\).

\( ab, bb \) Arrays:

\( ab \) (size at least max(1, \( ldab \times n \)) for column major layout and max(1, \( ldab \times (ka + 1) \)) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix \( A \) (as specified by \( uplo \)) in band storage format.

\( bb \) (size at least max(1, \( ldbb \times n \)) for column major layout and max(1, \( ldbb \times (kb + 1) \)) for row major layout) is an array containing either upper or lower triangular part of the symmetric matrix \( B \) (as specified by \( uplo \)) in band storage format.

\( ldab \) The leading dimension of the array \( ab \); must be at least \( ka+1 \) for column major layout and at least max(1, \( n \)) for row major layout.

\( ldbb \) The leading dimension of the array \( bb \); must be at least \( kb+1 \) for column major layout and at least max(1, \( n \)) for row major layout.

\( vl, vu \) If range = 'V', the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If range = 'A' or 'I', \( vl \) and \( vu \) are not referenced.

\( il, iu \) If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: 1 \( \leq il \leq iu \leq n \), if \( n > 0 \); \( il=1 \) and \( iu=0 \) if \( n = 0 \).

If range = 'A' or 'V', \( il \) and \( iu \) are not referenced.

\( abstol \) The absolute error tolerance for the eigenvalues. See Application Notes for more information.

\( ldz \) The leading dimension of the output array \( z \); \( ldz \geq 1 \). If jobz = 'V', \( ldz \geq \max(1, n) \).

\( ldq \) The leading dimension of the output array \( q \); \( ldq < 1 \).
If $jobz = 'V'$, $ldq < \max(1, n)$.

**Output Parameters**

**ab**

On exit, the contents of $ab$ are overwritten.

**bb**

On exit, contains the factor $S$ from the split Cholesky factorization $B = S^T*S$, as returned by pbstf/pbstf.

**m**

The total number of eigenvalues found,$0 \leq m \leq n$. If $range = 'A'$, $m = n$, and if $range = 'I'$, $m = il-1+iu$.

**w, z, q**

Arrays:

- $w$, size at least $\max(1, n)$.
  - If $info = 0$, contains the eigenvalues in ascending order.
  - If $jobz = 'V'$, then $z$ contains the matrix $Z$ of eigenvectors, with the $i$-th column of $z$ holding the eigenvector associated with $w(i)$. The eigenvectors are normalized so that $Z^T*B*Z = I$.
  - If $jobz = 'N'$, then $z$ is not referenced.

- $q$ (size $\max(1, ldq*n)$).
  - If $jobz = 'V'$, then $q$ contains the $n$-by-$n$ matrix used in the reduction of $A*x = \lambda*B*x$ to standard form, that is, $C*x = \lambda*B*x$ and consequently $C$ to tridiagonal form.
  - If $jobz = 'N'$, then $q$ is not referenced.

**ifail**

Array, size $m$.

- If $jobz = 'V'$, then if $info = 0$, the first $m$ elements of $ifail$ are zero; if $info > 0$, the $ifail$ contains the indices of the eigenvectors that failed to converge.
- If $jobz = 'N'$, then $ifail$ is not referenced.

**Return Values**

This function returns a value $info$.

- If $info=0$, the execution is successful.
- If $info = -i$, the $i$-th parameter had an illegal value.
- If $info > 0$, and
  - if $i \leq n$, the algorithm failed to converge, and $i$ off-diagonal elements of an intermediate tridiagonal did not converge to zero;
  - if $info = n + i$, for $1 \leq i \leq n$, then pbstf/pbstf returned $info = i$ and $B$ is not positive-definite. The factorization of $B$ could not be completed and no eigenvalues or eigenvectors were computed.
Application Notes

An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a,b]\) of width less than or equal to \(abstol+\varepsilon\max(|a|,|b|)\), where \(\varepsilon\) is the machine precision.

If \(abstol\) is less than or equal to zero, then \(\varepsilon\cdot||T||_1\) is used as tolerance, where \(T\) is the tridiagonal matrix obtained by reducing \(A\) to tridiagonal form. Eigenvalues will be computed most accurately when \(abstol\) is set to twice the underflow threshold \(2\cdot\text{lamch('S')}\), not zero.

If this routine returns with \(info > 0\), indicating that some eigenvectors did not converge, try setting \(abstol\) to \(2\cdot\text{lamch('S')}\).

\(\text{?hbgvx}\)

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem with banded matrices.

Syntax

```c
lapack_int LAPACKE_chbgvx( int matrix_layout, char jobz, char range, char uplo,
                        lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_float* ab, lapack_int ldab,
                        lapack_complex_float* bb, lapack_int ldbb, lapack_complex_float* q, lapack_int ldq,
                        float vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int* m, float* w,
                        lapack_complex_float* z, lapack_int ldz, lapack_int* ifail);

lapack_int LAPACKE_zhbgvx( int matrix_layout, char jobz, char range, char uplo,
                        lapack_int n, lapack_int ka, lapack_int kb, lapack_complex_double* ab, lapack_int ldab,
                        lapack_complex_double* bb, lapack_int ldbb, lapack_complex_double* q, lapack_int ldq,
                        double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int* m, double* w,
                        lapack_complex_double* z, lapack_int ldz, lapack_int* ifail );
```

Include Files

- mkl.h

Description

The routine computes selected eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite banded eigenproblem, of the form \(A^*x = \lambda^*B^*x\). Here \(A\) and \(B\) are assumed to be Hermitian and banded, and \(B\) is also positive definite. Eigenvalues and eigenvectors can be selected by specifying either all eigenvalues, a range of values or a range of indices for the desired eigenvalues.

Input Parameters

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `jobz` Must be 'N' or 'V'.
  - If `jobz` = 'N', then compute eigenvalues only.
  - If `jobz` = 'V', then compute eigenvalues and eigenvectors.
- `range` Must be 'A' or 'V' or 'I'.
  - If `range` = 'A', the routine computes all eigenvalues.
  - If `range` = 'V', the routine computes eigenvalues \(\omega[k]\) in the half-open interval:
If \( \text{range} = 'I' \), the routine computes eigenvalues with indices \( il \) to \( iu \).

**uplo**

Must be 'U' or 'L'.

If \( \text{uplo} = 'U' \), arrays \( ab \) and \( bb \) store the upper triangles of \( A \) and \( B \);

If \( \text{uplo} = 'L' \), arrays \( ab \) and \( bb \) store the lower triangles of \( A \) and \( B \).

**n**

The order of the matrices \( A \) and \( B \) (\( n \geq 0 \)).

**ka**

The number of super- or sub-diagonals in \( A \) (\( ka \geq 0 \)).

**kb**

The number of super- or sub-diagonals in \( B \) (\( kb \geq 0 \)).

**ab, bb**

Arrays:

\( ab \) (size at least \( \max(1, \text{ldab} \times n) \) for column major layout and \( \max(1, \text{ldab} \times (ka + 1)) \) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix \( A \) (as specified by \( \text{uplo} \)) in band storage format.

\( bb \) (size at least \( \max(1, \text{ldbb} \times n) \) for column major layout and \( \max(1, \text{ldbb} \times (kb + 1)) \) for row major layout) is an array containing either upper or lower triangular part of the Hermitian matrix \( B \) (as specified by \( \text{uplo} \)) in band storage format.

**ldab**

The leading dimension of the array \( ab \); must be at least \( ka + 1 \) for column major layout and at least \( \max(1, n) \) for row major layout.

**ldbb**

The leading dimension of the array \( bb \); must be at least \( kb + 1 \) for column major layout and at least \( \max(1, n) \) for row major layout.

**vl, vu**

If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues.

Constraint: \( vl < vu \).

If \( \text{range} = 'A' \) or 'I', \( vl \) and \( vu \) are not referenced.

**il, iu**

If \( \text{range} = 'I' \), the indices in ascending order of the smallest and largest eigenvalues to be returned.

Constraint: \( 1 \leq il \leq iu \leq n \), if \( n > 0 \); \( il = 1 \) and \( iu = 0 \) if \( n = 0 \).

If \( \text{range} = 'A' \) or 'V', \( il \) and \( iu \) are not referenced.

**abstol**

The absolute error tolerance for the eigenvalues. See Application Notes for more information.

**ldz**

The leading dimension of the output array \( z \); \( ldz \geq 1 \). If \( \text{jobz} = 'V' \), \( ldz \geq \max(1, n) \) for column major layout and at least \( \max(1, n) \) for row major layout.

**ldq**

The leading dimension of the output array \( q \); \( ldq \geq 1 \). If \( \text{jobz} = 'V' \), \( ldq \geq \max(1, n) \).
Output Parameters

**ab**
On exit, the contents of *ab* are overwritten.

**bb**
On exit, contains the factor *S* from the split Cholesky factorization *B* = *S*H*S*, as returned by **pbstf**.

**m**
The total number of eigenvalues found,
0 ≤ *m* ≤ *n*. If **range** = 'A', *m* = *n*, and if **range** = 'I',
m = *iu*-il+1.

**w**
Array *w*, size at least max(1, *n*).
If **info** = 0, contains the eigenvalues in ascending order.

**z, q**
Arrays:

*z* (size max(1, ldz* *m*) for column major layout and max(1, ldz* *n*) for row major layout).

If **jobz** = 'V', then if **info** = 0, *z* contains the matrix *Z* of eigenvectors, with the *i*-th column of *z* holding the eigenvector associated with *w*[ *i* - 1]. The eigenvectors are normalized so that *Z*H *B*H *Z* = I.

If **jobz** = 'N', then *z* is not referenced.

*q* (size max(1, ldq* *n*)).

If **jobz** = 'V', then *q* contains the *n*-by-*n* matrix used in the reduction of

*Ax* = *λBx* to standard form, that is, *Cx* = *λx* and consequently *C* to tridiagonal form.

If **jobz** = 'N', then *q* is not referenced.

**ifail**
Array, size at least max(1, *n*).

If **jobz** = 'V', then if **info** = 0, the first *m* elements of *ifail* are zero; if **info** > 0, the *ifail* contains the indices of the eigenvectors that failed to converge.

If **jobz** = 'N', then *ifail* is not referenced.

Return Values
This function returns a value **info**.

If **info** = 0, the execution is successful.

If **info** = *i*, the *i*-th parameter had an illegal value.

If **info** > 0, and

if **info** ≤ *n*, the algorithm failed to converge, and *i* off-diagonal elements of an intermediate tridiagonal did not converge to zero;

if **info** = *n* + *i*, for 1 ≤ *i* ≤ *n*, then **pbstf** returned **info** = *i* and *B* is not positive-definite. The factorization of *B* could not be completed and no eigenvalues or eigenvectors were computed.

Application Notes
An approximate eigenvalue is accepted as converged when it is determined to lie in an interval [a,b] of width
less than or equal to **abstol**+ε*max(|a|,|b|), where ε is the machine precision.
If $abstol$ is less than or equal to zero, then $\epsilon||T||_1$ will be used in its place, where $T$ is the tridiagonal matrix obtained by reducing $A$ to tridiagonal form. Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2*:\text{lamch}(\text{'S'})$, not zero.

If this routine returns with $info > 0$, indicating that some eigenvectors did not converge, try setting $abstol$ to $2*:\text{lamch}(\text{'S'})$.

**Generalized Nonsymmetric Eigenvalue Problems: LAPACK Driver Routines**

This section describes LAPACK driver routines used for solving generalized nonsymmetric eigenproblems. See also computational routines that can be called to solve these problems. Table "Driver Routines for Solving Generalized Nonsymmetric Eigenproblems" lists all such driver routines.

**Driver Routines for Solving Generalized Nonsymmetric Eigenproblems**

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gges</td>
<td>Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors for a pair of nonsymmetric matrices.</td>
</tr>
<tr>
<td>ggesx</td>
<td>Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right Schur vectors.</td>
</tr>
<tr>
<td>gges3</td>
<td>Computes generalized Schur factorization for a pair of matrices.</td>
</tr>
<tr>
<td>ggev</td>
<td>Computes the generalized eigenvalues, Schur form, and the left and/or right Schur vectors.</td>
</tr>
<tr>
<td>ggevx</td>
<td>Computes the generalized eigenvalues, and, optionally, the left and/or right Schur vectors.</td>
</tr>
<tr>
<td>ggev3</td>
<td>Computes generalized Schur factorization for a pair of matrices.</td>
</tr>
</tbody>
</table>

**Syntax**

```
lapack_int LAPACKE_sgges( int matrix_layout, char jobvsl, char jobvsr, char sort, 
LAPACK_S_SELECT3 select, lapack_int n, float* a, lapack_int lda, float* b, lapack_int 
ldb, lapack_int* sdim, float* alphar, float* alphai, float* beta, float* vsl, lapack_int 
ldvsl, float* vsr, lapack_int ldvsr );
```

```
lapack_int LAPACKE_dgges( int matrix_layout, char jobvsl, char jobvsr, char sort, 
LAPACK_D_SELECT3 select, lapack_int n, double* a, lapack_int lda, double* b, lapack_int 
ldb, lapack_int* sdim, double* alphar, double* alphai, double* beta, double* vsl, lapack_int 
ldvsl, double* vsr, lapack_int ldvsr );
```

```
lapack_int LAPACKE_cgges( int matrix_layout, char jobvsl, char jobvsr, char sort, 
LAPACK_C_SELECT2 select, lapack_int n, lapack_complex_float* a, lapack_int lda, 
lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float* alpha, 
lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl, 
lapack_complex_float* vsr, lapack_int ldvsr );
```

```
lapack_int LAPACKE_zgges( int matrix_layout, char jobvsl, char jobvsr, char sort, 
LAPACK_Z_SELECT2 select, lapack_int n, lapack_complex_double* a, lapack_int lda, 
lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double* alpha, 
lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl, 
lapack_complex_double* vsr, lapack_int ldvsr );
```
Include Files

- mkl.h

Description

The \texttt{?gges} routine computes the generalized eigenvalues, the generalized real/complex Schur form \((S,T)\), optionally, the left and/or right matrices of Schur vectors \((vsl\) and \(vsr\)) for a pair of \(n\)-by-\(n\) real/complex nonsymmetric matrices \((A,B)\). This gives the generalized Schur factorization

\[
(A,B) = (vsl*S*vsl^H, vsl*T*vsl^H)
\]

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix \(S\) and the upper triangular matrix \(T\). The leading columns of \(vsl\) and \(vsr\) then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

If only the generalized eigenvalues are needed, use the driver \texttt{ggev} instead, which is faster.

A generalized eigenvalue for a pair of matrices \((A,B)\) is a scalar \(w\) or a ratio \(\alpha / \beta = w\), such that \(A - w*B\) is singular. It is usually represented as the pair \((\alpha, \beta)\), as there is a reasonable interpretation for \(\beta=0\) or for both being zero. A pair of matrices \((S,T)\) is in the generalized real Schur form if \(T\) is upper triangular with non-negative diagonal and \(S\) is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of \(S\) are "standardized" by making the corresponding elements of \(T\) have the form:

\[
\begin{pmatrix}
\alpha & 0 \\
0 & \beta
\end{pmatrix}
\]

and the pair of corresponding 2-by-2 blocks in \(S\) and \(T\) will have a complex conjugate pair of generalized eigenvalues. A pair of matrices \((S,T)\) is in generalized complex Schur form if \(S\) and \(T\) are upper triangular and, in addition, the diagonal of \(T\) are non-negative real numbers.

The \texttt{?gges} routine replaces the deprecated \texttt{?gegs} routine.

Input Parameters

\texttt{matrix_layout}

Specifies whether matrix storage layout is row major (LAPACK\_ROW\_MAJOR) or column major (LAPACK\_COL\_MAJOR).

\texttt{jobvsl}

Must be \texttt{'N'} or \texttt{'V'}.

- If \texttt{jobvsl = 'N'}, then the left Schur vectors are not computed.
- If \texttt{jobvsl = 'V'}, then the left Schur vectors are computed.

\texttt{jobvsr}

Must be \texttt{'N'} or \texttt{'V'}.

- If \texttt{jobvsr = 'N'}, then the right Schur vectors are not computed.
- If \texttt{jobvsr = 'V'}, then the right Schur vectors are computed.

\texttt{sort}

Must be \texttt{'N'} or \texttt{'S'}. Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.

- If \texttt{sort = 'N'}, then eigenvalues are not ordered.
- If \texttt{sort = 'S'}, eigenvalues are ordered (see \texttt{select}).
The `select` parameter is a pointer to a function returning a value of `lapack_logical` type. For different flavors the function has different arguments:

- **LAPACKE_sgges**: `lapack_logical (*LAPACK_S_SELECT3) (const float*, const float*, const float* );`
- **LAPACKE_dgges**: `lapack_logical (*LAPACK_D_SELECT3) (const double*, const double*, const double* );`
- **LAPACKE_cgges**: `lapack_logical (*LAPACK_C_SELECT2) (const lapack_complex_float*, const lapack_complex_float* );`
- **LAPACKE_zgges**: `lapack_logical (*LAPACK_Z_SELECT2) (const lapack_complex_double*, const lapack_complex_double* );`

If `sort = 'S'`, `select` is used to select eigenvalues to sort to the top left of the Schur form.

If `sort = 'N'`, `select` is not referenced.

For real flavors:

An eigenvalue `(alphar[j] + alphai[j])/beta[j]` is selected if `select(alphar[j], alphai[j], beta[j])` is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy `select(alphar[j], alphai[j], beta[j]) = 1` after ordering. In this case `info` is set to `n+2`.

For complex flavors:

An eigenvalue `alpha[j] / beta[j]` is selected if `select(alpha[j], beta[j])` is true.

Note that a selected complex eigenvalue may no longer satisfy `select(alpha[j], beta[j]) = 1` after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case `info` is set to `n+2` (see `info` below).

The order of the matrices `A`, `B`, `vsl`, and `vsr` (`n≥ 0`).

**Arrays:**

- `a` (size at least `max(1, lda*n)`) is an array containing the `n`-by-`n` matrix `A` (first of the pair of matrices).
- `b` (size at least `max(1, ldb*n)`) is an array containing the `n`-by-`n` matrix `B` (second of the pair of matrices).

The leading dimension of the array `a`. Must be at least `max(1, n)`.

The leading dimension of the array `b`. Must be at least `max(1, n)`.

The leading dimensions of the output matrices `vsl` and `vsr`, respectively. Constraints:

- `ldvs1≥ 1. If jobvsl = 'V', ldvs1≥ max(1, n)`.
- `ldvsr≥ 1. If jobvsr = 'V', ldvsr≥ max(1, n)`.
Output Parameters

- **a**: On exit, this array has been overwritten by its generalized Schur form $S$.

- **b**: On exit, this array has been overwritten by its generalized Schur form $T$.

- **sdim**: If $\text{sort} = 'N'$, $sdim = 0$.

  If $\text{sort} = 'S'$, $sdim$ is equal to the number of eigenvalues (after sorting) for which $\text{select}$ is true.

  Note that for real flavors complex conjugate pairs for which $\text{select}$ is true for either eigenvalue count as 2.

- **alphar, alphai**: Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

  See $\beta$.

- **alpha**: Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See $\beta$.

- **beta**: Array, size at least $\max(1, n)$.

  For real flavors:

  On exit, $(\text{alphar}[j] + \text{alphai}[j]i)/\text{beta}[j], j=0,\ldots, n-1$, will be the generalized eigenvalues.

  $\text{alphar}[j] + \text{alphai}[j]i$ and $\text{beta}[j], j=0,\ldots, n-1$ are the diagonals of the complex Schur form $(S, T)$ that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of $(A, B)$ were further reduced to triangular form using complex unitary transformations. If alphai[j] is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with alphai[j+1] negative.

  For complex flavors:

  On exit, $\text{alpha}[j]/\text{beta}[j], j=0,\ldots, n-1$, will be the generalized eigenvalues. $\text{alpha}[j]$ and $\text{beta}[j], j=0,\ldots, n-1$ are the diagonals of the complex Schur form $(S, T)$ output by cgges/zgges. The $\text{beta}[j]$ will be non-negative real.

  See also Application Notes below.

- **vsl, vsr**: Arrays:

  $vsl$ (size at least $\max(1, ldvsl*n)$).

  If $\text{jobvsl} = 'V'$, this array will contain the left Schur vectors.

  If $\text{jobvsl} = 'N'$, $vsl$ is not referenced.

  $vsr$ (size at least $\max(1, ldvsr*n)$).

  If $\text{jobvsr} = 'V'$, this array will contain the right Schur vectors.

  If $\text{jobvsr} = 'N'$, $vsr$ is not referenced.

Return Values

This function returns a value $\text{info}$.

If $\text{info} = 0$, the execution is successful.

If $\text{info} = -i$, the $i$-th parameter had an illegal value.
If info = i, and 

\( i \leq n \):

the QZ iteration failed. \((A, B)\) is not in Schur form, but \(\alpha_{l, j}, \alpha_{r, j}\) (for real flavors), or \(\alpha_{j}\) (for complex flavors), and \(\beta_{j}, j = info, \ldots, n - 1\) should be correct.

\( i > n \): errors that usually indicate LAPACK problems:

\( i = n+1 \): other than QZ iteration failed in \(rgeqz\);

\( i = n+2 \): after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy \(\text{select} = 1\). This could also be caused due to scaling;

\( i = n+3 \): reordering failed in \(tgsen\).

**Application Notes**

The quotients \(\alpha_{l, j}/\beta_{j}\) and \(\alpha_{r, j}/\beta_{j}\) may easily over- or underflow, and \(\beta_{j}\) may even be zero. Thus, you should avoid simply computing the ratio. However, \(\alpha_{l}\) and \(\alpha_{r}\) will be always less than and usually comparable with \(\text{norm}(A)\) in magnitude, and \(\beta\) always less than and usually comparable with \(\text{norm}(B)\).

**?ggesx**

Computes the generalized eigenvalues, Schur form, and, optionally, the left and/or right matrices of Schur vectors.

**Syntax**

```c
lapack_int LAPACK_sggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_S_SELECT3 select, char sense, lapack_int n, float* a, lapack_int lda, float* b,
lapack_int ldb, lapack_int* sdim, float* alpha, float* alphai, float* beta, float* vsl,
lapack_int ldvsl, float* vsr, lapack_int ldvsl, float* rconde, float* rcondv );
```

```c
lapack_int LAPACK_dggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_D_SELECT3 select, char sense, lapack_int n, double* a, lapack_int lda, double* b,
lapack_int ldb, lapack_int* sdim, double* alpha, double* alphai, double* beta, double* vsl,
lapack_int ldvsl, double* vsr, lapack_int ldvsl, double* rconde, double* rcondv );
```

```c
lapack_int LAPACK_cggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 select, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda,
lapack_complex_float* b, lapack_int ldb, lapack_int* sdim, lapack_complex_float* alpha,
lapack_complex_float* beta, lapack_complex_float* vsl, lapack_int ldvsl,
lapack_complex_float* vsr, lapack_int ldvsl, float* rconde, float* rcondv );
```

```c
lapack_int LAPACK_zggesx( int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 select, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda,
lapack_complex_double* b, lapack_int ldb, lapack_int* sdim, lapack_complex_double* alpha,
lapack_complex_double* beta, lapack_complex_double* vsl, lapack_int ldvsl,
lapack_complex_double* vsr, lapack_int ldvsl, double* rconde, double* rcondv );
```

**Include Files**

- mkl.h

**Description**
The routine computes for a pair of \( n \times n \) real/complex nonsymmetric matrices \((A, B)\), the generalized eigenvalues, the generalized real/complex Schur form \((S, T)\), optionally, the left and/or right matrices of Schur vectors \( (vsl \text{ and } vsr) \). This gives the generalized Schur factorization

\[
(A, B) = (vsl \cdot S \cdot vsr^H, \; vsl \cdot T \cdot vsr^H)
\]

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix \( S \) and the upper triangular matrix \( T \); computes a reciprocal condition number for the average of the selected eigenvalues \((rconde)\); and computes a reciprocal condition number for the right and left deflating subspaces corresponding to the selected eigenvalues \((rcondv)\). The leading columns of \( vsl \) and \( vsr \) then form an orthonormal/unitary basis for the corresponding left and right eigenspaces (deflating subspaces).

A generalized eigenvalue for a pair of matrices \((A, B)\) is a scalar \( w \) or a ratio \( \alpha / \beta = w \), such that \( A - w \cdot B \) is singular. It is usually represented as the pair \((\alpha, \beta)\), as there is a reasonable interpretation for \( \beta = 0 \) or for both being zero. A pair of matrices \((S, T)\) is in generalized real Schur form if \( T \) is upper triangular with non-negative diagonal and \( S \) is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of \( S \) will be "standardized" by making the corresponding elements of \( T \) have the form:

\[
\begin{pmatrix}
\alpha & 0 \\
0 & \beta
\end{pmatrix}
\]

and the pair of corresponding 2-by-2 blocks in \( S \) and \( T \) will have a complex conjugate pair of generalized eigenvalues. A pair of matrices \((S, T)\) is in generalized complex Schur form if \( S \) and \( T \) are upper triangular and, in addition, the diagonal of \( T \) are non-negative real numbers.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major \((\text{LAPACK\_ROW\_MAJOR})\) or column major \((\text{LAPACK\_COL\_MAJOR})\).

- **jobvsl**
  Must be \('N' \text{ or } 'V'\).
  - If \( \text{jobvsl} = 'N' \), then the left Schur vectors are not computed.
  - If \( \text{jobvsl} = 'V' \), then the left Schur vectors are computed.

- **jobvsr**
  Must be \('N' \text{ or } 'V'\).
  - If \( \text{jobvsr} = 'N' \), then the right Schur vectors are not computed.
  - If \( \text{jobvsr} = 'V' \), then the right Schur vectors are computed.

- **sort**
  Must be \('N' \text{ or } 'S'\). Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.
  - If \( \text{sort} = 'N' \), then eigenvalues are not ordered.
  - If \( \text{sort} = 'S' \), eigenvalues are ordered (see \( \text{select} \)).

- **select**
  The \( \text{select} \) parameter is a pointer to a function returning a value of \( \text{lapack\_logical} \) type. For different flavors the function has different arguments:
  - \( \text{LAPACKE\_sggesx}: \text{lapack\_logical} \left( \ast \text{LAPACK\_S\_SELECT3} \right) \left( \text{const float*, const float*, const float* } \right) \);
  - \( \text{LAPACKE\_dggesx}: \text{lapack\_logical} \left( \ast \text{LAPACK\_D\_SELECT3} \right) \left( \text{const double*, const double*, const double* } \right) \);
LAPACKE_cggesx: `lapack_logical (*LAPACK_C_SELECT2) (const lapack_complex_float*, const lapack_complex_float* );`

LAPACKE_zggesx: `lapack_logical (*LAPACK_Z_SELECT2) (const lapack_complex_double*, const lapack_complex_double* );`

If `sort = 'S'`, `select` is used to select eigenvalues to sort to the top left of the Schur form.

If `sort = 'N'`, `select` is not referenced.

For real flavors:

An eigenvalue \( (\text{alphar}[j] + \text{alphai}[j])/\text{beta}[j] \) is selected if `select(\text{alphar}[j], \text{alphai}[j], \text{beta}[j])` is true; that is, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy `select(\text{alphar}[j], \text{alphai}[j], \text{beta}[j]) = 1` after ordering. In this case `info` is set to `n+2`.

For complex flavors:

An eigenvalue \( \alpha[j]/\beta[j] \) is selected if `select(\alpha[j], \beta[j])` is true.

Note that a selected complex eigenvalue may no longer satisfy `select(\alpha[j], \beta[j]) = 1` after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned); in this case `info` is set to `n+2` (see `info` below).

**sense**

Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If `sense = 'N'`, none are computed;

If `sense = 'E'`, computed for average of selected eigenvalues only;

If `sense = 'V'`, computed for selected deflating subspaces only;

If `sense = 'B'`, computed for both.

If `sense` is 'E', 'V', or 'B', then `sort` must equal 'S'.

**n**

The order of the matrices \( A, B, vsl, \) and \( vsr (n \geq 0) \).

**a, b**

Arrays:

\( a \) (size at least max(1, lda*n)) is an array containing the \( n \)-by-\( n \) matrix \( A \) (first of the pair of matrices).

\( b \) (size at least max(1, ldb*n)) is an array containing the \( n \)-by-\( n \) matrix \( B \) (second of the pair of matrices).

**lda**

The leading dimension of the array \( a \).

Must be at least max(1, n).

**ldb**

The leading dimension of the array \( b \).

Must be at least max(1, n).

**ldvsl, ldvsr**

The leading dimensions of the output matrices \( vsl \) and \( vsr \), respectively.

Constraints:
If $\text{jobvsl} = 'V'$, $\text{ldvsl} \geq \max(1, n)$.

If $\text{jobvsr} = 'V'$, $\text{ldvsr} \geq \max(1, n)$.

Output Parameters

- **a**: On exit, this array has been overwritten by its generalized Schur form $S$.

- **b**: On exit, this array has been overwritten by its generalized Schur form $T$.

- **sdim**: If $\text{sort} = 'N'$, $\text{sdim} = 0$.

  If $\text{sort} = 'S'$, $\text{sdim}$ is equal to the number of eigenvalues (after sorting) for which $\text{select}$ is true.

  Note that for real flavors complex conjugate pairs for which $\text{select}$ is true for either eigenvalue count as 2.

- **alphar, alphai**: Arrays, size at least $\max(1, n)$ each. Contain values that form generalized eigenvalues in real flavors.

  See $\beta$.

- **alpha**: Array, size at least $\max(1, n)$. Contain values that form generalized eigenvalues in complex flavors. See $\beta$.

- **beta**: Array, size at least $\max(1, n)$.

  For real flavors:

  On exit, $\left(\text{alphar}[j] + \text{alphai}[j] \ast i\right) / \text{beta}[j], j=0,\ldots, n-1$ will be the generalized eigenvalues.

  $\text{alphar}[j] + \text{alphai}[j] \ast i$ and $\text{beta}[j], j=0,\ldots, n-1$ are the diagonals of the complex Schur form $(S,T)$ that would result if the 2-by-2 diagonal blocks of the real generalized Schur form of $(A,B)$ were further reduced to triangular form using complex unitary transformations. If $\text{alphai}[j]$ is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alphai}[j+1]$ negative.

  For complex flavors:

  On exit, $\text{alpha}[j]/\text{beta}[j], j=0,\ldots, n-1$ will be the generalized eigenvalues. $\text{alpha}[j]$ and $\text{beta}[j], j=0,\ldots, n-1$ are the diagonals of the complex Schur form $(S,T)$ output by $\text{cggesx/zggesx}$. The $\text{beta}[j]$ will be non-negative real.

  See also Application Notes below.

- **vsl, vsr**: Arrays:

  $\text{vsl}$ (size at least $\max(1, \text{ldvsl} \ast n)$).

  If $\text{jobvsl} = 'V'$, this array will contain the left Schur vectors.

  If $\text{jobvsl} = 'N'$, $\text{vsl}$ is not referenced.

  $\text{vsr}$ (size at least $\max(1, \text{ldvsr} \ast n)$).

  If $\text{jobvsr} = 'V'$, this array will contain the right Schur vectors.

  If $\text{jobvsr} = 'N'$, $\text{vsr}$ is not referenced.

- **rconde, rcondv**: Arrays, size 2 each
If sense = 'E' or 'B', rconde(1) and rconde(2) contain the reciprocal condition numbers for the average of the selected eigenvalues.
Not referenced if sense = 'N' or 'V'.
If sense = 'V' or 'B', rcondv[0] and rcondv[1] contain the reciprocal condition numbers for the selected deflating subspaces.
Not referenced if sense = 'N' or 'E'.

Return Values

This function returns a value info.
If info=0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.
If info = i, and
i≤n:
the QZ iteration failed. (A, B) is not in Schur form, but alphar[j], alphai[j] (for real flavors), or alpha[j] (for complex flavors), and beta[j], j = info, ..., n - 1 should be correct.
i > n: errors that usually indicate LAPACK problems:
i = n+1: other than QZ iteration failed in hgeqz;
i = n+2: after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the generalized Schur form no longer satisfy select = 1. This could also be caused due to scaling;
i = n+3: reordering failed in tgsen.

Application Notes

The quotients alphar[j]/beta[j] and alphai[j]/beta[j] may easily over- or underflow, and beta[j] may even be zero. Thus, you should avoid simply computing the ratio. However, alphar and alphai will be always less than and usually comparable with norm(A) in magnitude, and beta always less than and usually comparable with norm(B).

?qgges3
Computes generalized Schur factorization for a pair of matrices.

Syntax

```c
lapack_int LAPACKE_sgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort, LAPACK_S_SELECT3 selctg, lapack_int n, float * a, lapack_int lda, float * b, lapack_int ldb, lapack_int * sdim, float * alphar, float * alphai, float * beta, float * vsl, lapack_int ldvsl, float * vsr, lapack_int ldvsr);

lapack_int LAPACKE_dgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort, LAPACK_D_SELECT3 selctg, lapack_int n, double * a, lapack_int lda, double * b, lapack_int ldb, lapack_int * sdim, double * alphar, double * alphai, double * beta, double * vsl, lapack_int ldvsl, double * vsr, lapack_int ldvsr);
```
lapack_int LAPACKE_cggges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_C_SELECT2 selctg, lapack_int n, lapack_complex_float * a, lapack_int lda,
lapack_complex_float * b, lapack_int ldb, lapack_int * sdim, lapack_complex_float * alpha,
lapack_complex_float * beta, lapack_complex_float * vsl, lapack_int ldvsl,
lapack_complex_float * vsr, lapack_int ldvsr);

lapack_int LAPACKE_zgges3 (int matrix_layout, char jobvsl, char jobvsr, char sort,
LAPACK_Z_SELECT2 selctg, lapack_int n, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, lapack_int * sdim, lapack_complex_double * alpha,
lapack_complex_double * beta, lapack_complex_double * vsl, lapack_int ldvsl,
lapack_complex_double * vsr, lapack_int ldvsr);

Include Files
• mkl.h

Description
For a pair of \( n \times n \) real or complex nonsymmetric matrices (\( A,B \)), \( ?gges3 \) computes the generalized eigenvalues, the generalized real or complex Schur form (\( S,T \)), and optionally the left or right matrices of Schur vectors (\( VSL \) and \( VSR \)). This gives the generalized Schur factorization
\[
(A,B) = ( (VSL)*S*(VSR)^T, (VSL)*T*(VSR)^T ) \quad \text{for real } (A,B)
\]
or
\[
(A,B) = ( (VSL)*S*(VSR)^H, (VSL)*T*(VSR)^H ) \quad \text{for complex } (A,B)
\]
where \((VSR)^H\) is the conjugate-transpose of \( VSR \).

Optionally, it also orders the eigenvalues so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix \( S \) and the upper triangular matrix \( T \). The leading columns of \( VSL \) and \( VSR \) then form an orthonormal basis for the corresponding left and right eigenspaces (deflating subspaces).

NOTE
If only the generalized eigenvalues are needed, use the driver \( ?ggev \) instead, which is faster.

A generalized eigenvalue for a pair of matrices \((A,B)\) is a scalar \( w \) or a ratio \( \alpha/\beta = w \), such that \( A - w \cdot B \) is singular. It is usually represented as the pair \((\alpha,\beta)\), as there is a reasonable interpretation for \( \beta = 0 \) or both being zero.

For real flavors:
A pair of matrices \((S,T)\) is in generalized real Schur form if \( T \) is upper triangular with non-negative diagonal and \( S \) is block upper triangular with 1-by-1 and 2-by-2 blocks. 1-by-1 blocks correspond to real generalized eigenvalues, while 2-by-2 blocks of \( S \) will be "standardized" by making the corresponding elements of \( T \) have the form:
\[
\begin{pmatrix}
  a & 0 \\
  0 & b
\end{pmatrix}
\]
and the pair of corresponding 2-by-2 blocks in \( S \) and \( T \) have a complex conjugate pair of generalized eigenvalues.

For complex flavors:
A pair of matrices \((S,T)\) is in generalized complex Schur form if \( S \) and \( T \) are upper triangular and, in addition, the diagonal elements of \( T \) are non-negative real numbers.
**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (**LAPACK_ROW_MAJOR**) or column major (**LAPACK_COL_MAJOR**).

- **jobvsl**
  = 'N': do not compute the left Schur vectors;

- **jobvsr**
  = 'N': do not compute the right Schur vectors;
  = 'V': compute the right Schur vectors.

- **sort**
  Specifies whether or not to order the eigenvalues on the diagonal of the generalized Schur form.
  = 'N': Eigenvalues are not ordered;
  = 'S': Eigenvalues are ordered (see **selctg**).

- **selctg**
  * selctg is a function of three arguments for real flavors or two arguments for complex flavors. selctg must be declared EXTERNAL in the calling subroutine. If sort = 'N', selctg is not referenced. If sort = 'S', selctg is used to select eigenvalues to sort to the top left of the Schur form.

  For real flavors:
  An eigenvalue \((\text{alphar}[j - 1] + \text{alphai}[j - 1])/\text{beta}[j - 1]\) is selected if \(\text{selctg}(\text{alphar}[j - 1], \text{alphai}[j - 1], \text{beta}[j - 1])\) is true.

  In other words, if either one of a complex conjugate pair of eigenvalues is selected, then both complex eigenvalues are selected.

  Note that in the ill-conditioned case, a selected complex eigenvalue may no longer satisfy \(\text{selctg}(\text{alphar}[j - 1], \text{alphai}[j - 1], \text{beta}[j - 1]) \neq 0\) after ordering. info is to be set to \(n+2\) in this case.

  For complex flavors:
  An eigenvalue \(\text{alpha}[j - 1]/\text{beta}[j - 1]\) is selected if \(\text{selctg}(\text{alpha}[j - 1], \text{beta}[j - 1])\) is true.

  Note that a selected complex eigenvalue may no longer satisfy \(\text{selctg}(\text{alpha}[j - 1], \text{beta}[j - 1]) \neq 0\) after ordering, since ordering may change the value of complex eigenvalues (especially if the eigenvalue is ill-conditioned), in this case ?gges3 returns \(n + 2\).

- **n**
  The order of the matrices \(A, B, VSL,\) and \(VSR. n \geq 0\).

- **a**
  Array, size \((lda*n)\). On entry, the first of the pair of matrices.

- **lda**
  The leading dimension of \(a. lda\geq max(1,n)\).

- **b**
  Array, size \((ldb*n)\). On entry, the second of the pair of matrices.

- **ldb**
  The leading dimension of \(b. ldb\geq max(1,n)\).

- **ldvsl**
  The leading dimension of the matrix \(VSL. ldvsl\geq 1\), and if \(jobvsl = 'V'\), \(ldvsl\geq n\).

- **ldvsr**
  The leading dimension of the matrix \(VSR. ldvsr\geq 1\), and if \(jobvsr = 'V'\), \(ldvsr\geq n\).
Output Parameters

On exit, \( a \) is overwritten by its generalized Schur form \( S \).

On exit, \( b \) is overwritten by its generalized Schur form \( T \).

If \( sort = 'N' \), \( sdim = 0 \). If \( sort = 'S' \), \( sdim = \) number of eigenvalues (after sorting) for which \( selctg \) is true.

\( \alpha \)

Array, size \( (n) \).

\( \text{alphar} \)

Array, size \( (n) \).

\( \text{alphai} \)

Array, size \( (n) \).

\( \beta \)

Array, size \( (n) \).

For real flavors:

On exit, \( (\text{alphar}[j - 1] + \text{alphai}[j - 1]*i)/\beta[j - 1], j=1,...,n \), are the generalized eigenvalues. \( \text{alphar}[j - 1] + \text{alphai}[j - 1]*i, \) and \( \beta[j - 1], j=1,...,n \) are the diagonals of the complex Schur form \( (S,T) \) that would result if the 2-by-2 diagonal blocks of the real Schur form of \( (a,b) \) were further reduced to triangular form using 2-by-2 complex unitary transformations. If \( \text{alphai}[j - 1] \) is zero, then the \( j \)-th eigenvalue is real; if positive, then the \( j \)-th and \( (j+1) \)-st eigenvalues are a complex conjugate pair, with \( \text{alphai}[j] \) negative.

Note: the quotients \( \text{alphar}[j - 1]/\beta[j - 1] \) and \( \text{alphai}[j - 1]/\beta[j - 1] \) can easily over- or underflow, and \( \beta[j - 1] \) might even be zero. Thus, you should avoid computing the ratio \( \alpha/\beta \) by simply dividing \( \alpha \) by \( \beta \). However, \( \text{alphar} \) and \( \text{alphai} \) is always less than and usually comparable with \( \text{norm}(a) \) in magnitude, and \( \beta \) is always less than and usually comparable with \( \text{norm}(b) \).

For complex flavors:

On exit, \( \text{alpha}[j - 1][j - 1]/\beta[j - 1], j=1,...,n \), are the generalized eigenvalues. \( \text{alpha}[j - 1], j=1,...,n \) and \( \beta[j - 1], j=1,...,n \) are the diagonals of the complex Schur form \( (a,b) \) output by \( \texttt{?gges3} \). The \( \beta[j - 1] \) is non-negative real.

Note: the quotient \( \text{alpha}[j - 1]/\beta[j - 1] \) can easily over- or underflow, and \( \beta[j - 1] \) might even be zero. Thus, you should avoid computing the ratio \( \alpha/\beta \) by simply dividing \( \alpha \) by \( \beta \). However, \( \alpha \) is always less than and usually comparable with \( \text{norm}(a) \) in magnitude, and \( \beta \) is always less than and usually comparable with \( \text{norm}(b) \).

\( \text{vsl} \)

Array, size \( (ldvsl*n) \).

If \( jobvsl = 'V' \), \( vsl \) contains the left Schur vectors. Not referenced if \( jobvsl = 'N' \).

\( \text{vsr} \)

Array, size \( (ldvsr*n) \).
If \( \text{jobvsr} = 'V' \), \( \text{vsr} \) contains the right Schur vectors. Not referenced if \( \text{jobvsr} = 'N' \).

**Return Values**

This function returns a value \( \text{info} \).

- \( = 0 \): successful exit
- \( < 0 \): if \( \text{info} = -i \), the \( i \)-th argument had an illegal value.
- \( = 1, \ldots, n \):
  - for real flavors: The QZ iteration failed. \((a,b)\) are not in Schur form, but \( \text{alphar} \[j \], \text{alphai} \[j \] \) and \( \text{beta} \[j \] \) should be correct for \( j=\text{info},\ldots,n-1 \).
  - for complex flavors:
    - \( > n \): \( =n+1 \): other than QZ iteration failed in \( \text{?hgeqz} \).
    - \( =n+2 \): after reordering, roundoff changed values of some complex eigenvalues so that leading eigenvalues in the Generalized Schur form no longer satisfy \( \text{selctg} \neq 0 \) This could also be caused due to scaling.
    - \( =n+3 \): reordering failed in \( \text{?tgsen} \).

**?ggev**

Computes the generalized eigenvalues, and the left and/or right generalized eigenvectors for a pair of nonsymmetric matrices.

**Syntax**

```c
lapack_int LAPACKE_sggev( int matrix_layout, char jobvl, char jobvr, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* alphar, float* alphai, float* beta, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr );
```

```c
lapack_int LAPACKE_dggev( int matrix_layout, char jobvl, char jobvr, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* alphar, double* alphai, double* beta, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr );
```

```c
lapack_int LAPACKE_cggev( int matrix_layout, char jobvl, char jobvr, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb, lapack_complex_float* alpha, lapack_complex_float* beta, lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr );
```

```c
lapack_int LAPACKE_zggev( int matrix_layout, char jobvl, char jobvr, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb, lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr );
```

**Include Files**

- \text{mkl.h}

**Description**

The \( ?ggev \) routine computes the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors for a pair of \( n \)-by-\( n \) real/complex nonsymmetric matrices \((A,B)\).
A generalized eigenvalue for a pair of matrices \((A,B)\) is a scalar \(\lambda\) or a ratio \(\frac{\alpha}{\beta} = \lambda\), such that \(A - \lambda B\) is singular. It is usually represented as the pair \((\alpha, \beta)\), as there is a reasonable interpretation for \(\beta = 0\) and even for both being zero.

The right generalized eigenvector \(v(j)\) corresponding to the generalized eigenvalue \(\lambda(j)\) of \((A,B)\) satisfies
\[
A*v(j) = \lambda(j)*B*v(j).
\]

The left generalized eigenvector \(u(j)\) corresponding to the generalized eigenvalue \(\lambda(j)\) of \((A,B)\) satisfies
\[
u(j)^H*A = \lambda(j)*u(j)^H*B
\]
where \(u(j)^H\) denotes the conjugate transpose of \(u(j)\).

The \(?\text{ggev}\) routine replaces the deprecated \(?\text{gegv}\) routine.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobvl**
  Must be \('N'\) or \('V'\).
  - If \(\text{jobvl} = \text{'N'}\), the left generalized eigenvectors are not computed;
  - If \(\text{jobvl} = \text{'V'}\), the left generalized eigenvectors are computed.

- **jobvr**
  Must be \('N'\) or \('V'\).
  - If \(\text{jobvr} = \text{'N'}\), the right generalized eigenvectors are not computed;
  - If \(\text{jobvr} = \text{'V'}\), the right generalized eigenvectors are computed.

- **n**
  The order of the matrices \(A, B, vl, \text{and } vr\) \((n \geq 0)\).

- **a, b**
  Arrays:
  - \(a\) (size at least \(\max(1, lda*n)\)) is an array containing the \(n\)-by-\(n\) matrix \(A\) (first of the pair of matrices).
  - \(b\) (size at least \(\max(1, ldb*n)\)) is an array containing the \(n\)-by-\(n\) matrix \(B\) (second of the pair of matrices).

- **lda**
  The leading dimension of the array \(a\). Must be at least \(\max(1, n)\).

- **ldb**
  The leading dimension of the array \(b\). Must be at least \(\max(1, n)\).

- **ldvl, ldvr**
  The leading dimensions of the output matrices \(vl\) and \(vr\), respectively.

  Constraints:
  - \(ldvl \geq 1\). If \(\text{jobvl} = \text{'V'}\), \(ldvl \geq \max(1, n)\).
  - \(ldvr \geq 1\). If \(\text{jobvr} = \text{'V'}\), \(ldvr \geq \max(1, n)\).

**Output Parameters**

- **a, b**
  On exit, these arrays have been overwritten.

- **alphar, alphai**
  Arrays, size at least \(\max(1, n)\) each. Contain values that form generalized eigenvalues in real flavors.

  See \(\beta\).
alpha

Array, size at least max(1, n). Contain values that form generalized eigenvalues in complex flavors. See beta.

beta

Array, size at least max(1, n).

For real flavors:

On exit, $(\text{alphar}[j] + \text{alpha}[j]*i)/\text{beta}[j], j=0,..., n - 1$, are the generalized eigenvalues.

If $\text{alpha}[j]$ is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $\text{alpha}[j+1]$ negative.

For complex flavors:

On exit, $\alpha[j]/\beta[j], j=0,..., n - 1$, are the generalized eigenvalues.

See also Application Notes below.

$vl, vr$

$vl$ (size at least max(1, ldvl*n)). Contains the matrix of left generalized eigenvectors $VL$.

If $\text{jobvl} = 'V'$, the left generalized eigenvectors $u_j$ are stored one after another in the columns of $VL$, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs(Re)} + \text{abs(Im)} = 1$.

If $\text{jobvl} = 'N'$, $vl$ is not referenced.

For real flavors:

If the $j$-th eigenvalue is real, then the $k$-th component of the $j$-th left eigenvector $u_j$ is stored in $vl[(k - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(k - 1)*ldvl + (j - 1)]$ for row major layout.

If the $j$-th and $(j+1)$-st eigenvalues form a complex conjugate pair, then for $i = \sqrt{-1}$, the $k$-th components of the $j$-th left eigenvector $u_j$ are $vl[(k - 1) + (j - 1)*ldvl] + i*vl[(k - 1) + j*ldvl]$ for column major layout and $vl[(k - 1)*ldvl + (j - 1)] + i*vl[(k - 1)*ldvl + j]$ for row major layout.

For complex flavors:

The $k$-th component of the $j$-th left eigenvector $u_j$ is stored in $vl[(k - 1) + (j - 1)*ldvl]$ for column major layout and in $vl[(k - 1)*ldvl + (j - 1)]$ for row major layout.

$vr$ (size at least max(1, ldvr*n)). Contains the matrix of right generalized eigenvectors $VR$.

If $\text{jobvr} = 'V'$, the right generalized eigenvectors $v_j$ are stored one after another in the columns of $VR$, in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has $\text{abs(Re)} + \text{abs(Im)} = 1$.

If $\text{jobvr} = 'N'$, $vr$ is not referenced.

For real flavors:
If the \( j \)-th eigenvalue is real, then the \( k \)-th component of the \( j \)-th right eigenvector \( v_j \) is stored in \( vr[(k - 1) + (j - 1) \cdot ldvr] \) for column major layout and in \( vr[(k - 1) \cdot ldvr + (j - 1)] \) for row major layout.

If the \( j \)-th and \((j+1)\)-st eigenvalues form a complex conjugate pair, then the \( k \)-th components of the \( j \)-th right eigenvector \( v_j \) can be computed as \( vr[(k - 1) + (j - 1) \cdot ldvr] + i \cdot vr[(k - 1) + j \cdot ldvr] \) for column major layout and \( vr[(k - 1) \cdot ldvr + (j - 1)] + i \cdot vr[(k - 1) \cdot ldvr + j] \) for row major layout. Similarly, the \( k \)-th components of the right eigenvector \( v_{j+1} \) can be computed as \( vr[(k - 1) + (j - 1) \cdot ldvr] - i \cdot vr[(k - 1) + j \cdot ldvr] \) for column major layout and \( vr[(k - 1) \cdot ldvr + (j - 1)] - i \cdot vr[(k - 1) \cdot ldvr + j] \) for row major layout.

\textit{For complex flavors:}

The \( k \)-th component of the \( j \)-th right eigenvector \( v_j \) is stored in \( vr[(k - 1) + (j - 1) \cdot ldvr] \) for column major layout and in \( vr[(k - 1) \cdot ldvr + (j - 1)] \) for row major layout.

\textbf{Return Values}

This function returns a value \( \text{info} \).

If \( \text{info}=0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = i \), and

\( i \leq n \): the QZ iteration failed. No eigenvectors have been calculated, but \( \text{alphar}[j] \), \( \text{alphai}[j] \) (for real flavors), or \( \text{alpha}[j] \) (for complex flavors), and \( \text{beta}[j], j=\text{info}, \ldots, n-1 \) should be correct.

\( i > n \): errors that usually indicate LAPACK problems:

\( i = n+1 \): other than QZ iteration failed in \( \text{hgeqz} \);

\( i = n+2 \): error return from \( \text{tgevc} \).

\textbf{Application Notes}

The quotients \( \text{alphar}[j]/\text{beta}[j] \) and \( \text{alphai}[j]/\text{beta}[j] \) may easily over- or underflow, and \( \text{beta}[j] \) may even be zero. Thus, you should avoid simply computing the ratio. However, \( \text{alphar} \) and \( \text{alphai} \) (for real flavors) or \( \text{alpha} \) (for complex flavors) will be always less than and usually comparable with norm(\( A \)) in magnitude, and \( \text{beta} \) always less than and usually comparable with norm(\( B \)).

?ggevx

\textit{Computes the generalized eigenvalues, and, optionally, the left and/or right generalized eigenvectors.}

\textbf{Syntax}

\begin{verbatim}
lapack_int LAPACKE_sggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, float* a, lapack_int lda, float* b, lapack_int ldb, float* alphar, float* alphai, float* beta, float* vl, lapack_int ldvl, float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm, float* bbnrm, float* rconde, float* rcondv );
\end{verbatim}
The routine computes for a pair of eigenvalues, and optionally, the left and/or right generalized eigenvectors. Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ilo, ihi, float* lscale, float* rscale, float* abnrm, float* bbnrm, float* rconde, float* rcondv);

lapack_int LAPACKE_dggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, double* a, lapack_int lda, double* b, lapack_int ldb, double* alphar, double* alphai, double* beta, double* vl, lapack_int ldvl, double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale, double* abnrm, double* bbnrm, double* rconde, double* rcondv);

lapack_int LAPACKE_cggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldab, lapack_complex_float* alpha, lapack_complex_float* beta, lapack_complex_float* vl, lapack_int ldvl, lapack_complex_float* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, float* lscale, float* rscale, float* abnrm, float* bbnrm, float* rconde, float* rcondv);

lapack_int LAPACKE_zggevx( int matrix_layout, char balanc, char jobvl, char jobvr, char sense, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldab, lapack_complex_double* alpha, lapack_complex_double* beta, lapack_complex_double* vl, lapack_int ldvl, lapack_complex_double* vr, lapack_int ldvr, lapack_int* ilo, lapack_int* ihi, double* lscale, double* rscale, double* abnrm, double* bbnrm, double* rconde, double* rcondv);

Include Files
- mkl.h

Description
The routine computes for a pair of \( n \times n \) real/complex nonsymmetric matrices \((A,B)\), the generalized eigenvalues, and optionally, the left and/or right generalized eigenvectors. Optionally also, it computes a balancing transformation to improve the conditioning of the eigenvalues and eigenvectors (ilo, ihi, lscale, rscale, abnrm, and bbnrm), reciprocal condition numbers for the eigenvalues (rconde), and reciprocal condition numbers for the right eigenvectors (rcondv).

A generalized eigenvalue for a pair of matrices \((A,B)\) is a scalar \( \lambda \) or a ratio \( \alpha / \beta = \lambda \), such that \( A - \lambda B \) is singular. It is usually represented as the pair \((\alpha, \beta)\), as there is a reasonable interpretation for \( \beta = 0 \) and even for both being zero. The right generalized eigenvector \( v(j) \) corresponding to the generalized eigenvalue \( \lambda(j) \) of \((A,B)\) satisfies
\[
A v(j) = \lambda(j) B v(j).
\]
The left generalized eigenvector \( u(j) \) corresponding to the generalized eigenvalue \( \lambda(j) \) of \((A,B)\) satisfies
\[
u(j)^H A = \lambda(j) u(j)^H B
\]
where \( u(j)^H \) denotes the conjugate transpose of \( u(j) \).

Input Parameters

\textbf{matrix_layout} \hspace{1cm} Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

\textbf{balanc} \hspace{1cm} Must be 'N', 'P', 'S', or 'B'. Specifies the balance option to be performed.
- If \text{balanc} = 'N', do not diagonally scale or permute;
- If \text{balanc} = 'P', permute only;
- If \text{balanc} = 'S', scale only;
- If \text{balanc} = 'B', both permute and scale.
Computed reciprocal condition numbers will be for the matrices after balancing and/or permuting. Permuting does not change condition numbers (in exact arithmetic), but balancing does.

**jobvl**
Must be 'N' or 'V'.

If jobvl = 'N', the left generalized eigenvectors are not computed;
If jobvl = 'V', the left generalized eigenvectors are computed.

**jobvr**
Must be 'N' or 'V'.

If jobvr = 'N', the right generalized eigenvectors are not computed;
If jobvr = 'V', the right generalized eigenvectors are computed.

**sense**
Must be 'N', 'E', 'V', or 'B'. Determines which reciprocal condition number are computed.

If sense = 'N', none are computed;
If sense = 'E', computed for eigenvalues only;
If sense = 'V', computed for eigenvectors only;
If sense = 'B', computed for eigenvalues and eigenvectors.

**n**
The order of the matrices A, B, vl, and vr (n ≥ 0).

**a, b**
Arrays:

a (size at least max(1, lda*n)) is an array containing the n-by-n matrix A (first of the pair of matrices).

b (size at least max(1, ldb*n)) is an array containing the n-by-n matrix B (second of the pair of matrices).

**lda**
The leading dimension of the array a.
Must be at least max(1, n).

**ldb**
The leading dimension of the array b.
Must be at least max(1, n).

**ldvl, ldvr**
The leading dimensions of the output matrices vl and vr, respectively.

Constraints:

ldvl≥ 1. If jobvl = 'V', ldvl≥ max(1, n).
ldvr≥ 1. If jobvr = 'V', ldvr≥ max(1, n).

**Output Parameters**

**a, b**
On exit, these arrays have been overwritten.

If jobvl = 'V' or jobvr = 'V' or both, then a contains the first part of the real Schur form of the "balanced" versions of the input A and B, and b contains its second part.

**alphar, alphai**
Arrays, size at least max(1, n) each. Contain values that form generalized eigenvalues in real flavors.

See beta.
**alpha**

Array, size at least max(1, n). Contain values that form generalized eigenvalues in complex flavors. See **beta**.

**beta**

Array, size at least max(1, n).

---

**For real flavors:**

On exit, \((\text{alphar}[j] + \text{alphai}[j]*i)/\text{beta}[j], j=0,..., n - 1\), will be the generalized eigenvalues.

If \(\text{alphai}[j]\) is zero, then the \(j\)-th eigenvalue is real; if positive, then the \(j\)-th and \((j+1)\)-st eigenvalues are a complex conjugate pair, with \(\text{alphai}[j+1]\) negative.

**For complex flavors:**

On exit, \(\text{alphar}[j]/\text{beta}[j], j=0,..., n - 1\), will be the generalized eigenvalues.

See also Application Notes below.

---

**vl, vr**

Arrays:

\(vl\) (size at least max(1, ldvl*n)).

If \(jobvl = 'V'\), the left generalized eigenvectors \(u(j)\) are stored one after another in the columns of \(vl\), in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.

If \(jobvl = 'N'\), \(vl\) is not referenced.

**For real flavors:**

If the \(j\)-th eigenvalue is real, then \(k\)-th component of \(j\)-th left eigenvector \(u_j\) is stored in \(vl[(k - 1) + (j - 1)*ldvl]\) for column major layout and in \(vl[(k - 1)*ldvl + (j - 1)]\) for row major layout.

If the \(j\)-th and \((j+1)\)-st eigenvalues form a complex conjugate pair, then for \(i = \text{sqrt}(-1)\), the \(k\)-th components of the \(j\)-th left eigenvector \(u_j\) can be computed as \(vl[(k - 1) + (j - 1)*ldvl] + i*vl[(k - 1) + j*ldvl]\) for column major layout and \(vl[(k - 1)*ldvl + (j - 1)] + i*vl[(k - 1)*ldvl + j]\) for row major layout. Similarly, the \(k\)-th components of the left eigenvector \(u_{j+1}\) can be computed as \(vl[(k - 1) + (j - 1)*ldvl] - i*vl[(k - 1) + j*ldvl]\) for column major layout and \(vl[(k - 1)*ldvl + (j - 1)] - i*vl[(k - 1)*ldvl + j]\) for row major layout.

**For complex flavors:**

The \(k\)-th component of the \(j\)-th left eigenvector \(u_j\) is stored in \(vl[(k - 1) + (j - 1)*ldvl]\) for column major layout and in \(vl[(k - 1)*ldvl + (j - 1)]\) for row major layout.

\(vr\) (size at least max(1, ldvr*n)).

If \(jobvr = 'V'\), the right generalized eigenvectors \(v(j)\) are stored one after another in the columns of \(vr\), in the same order as their eigenvalues. Each eigenvector will be scaled so the largest component have abs(Re) + abs(Im) = 1.

If \(jobvr = 'N'\), \(vr\) is not referenced.

**For real flavors:**
If the j-th eigenvalue is real, then the k-th component of the j-th right eigenvector \( v_j \) is stored in \( vr[(k - 1) + (j - 1) * ldvr] \) for column major layout and in \( vr[(k - 1) * ldvr + (j - 1)] \) for row major layout.

If the j-th and (j+1)-st eigenvalues form a complex conjugate pair, then

The k-th components of the j-th right eigenvector \( v_j \) can be computed as

\[
vr[(k - 1) + (j - 1) * ldvr] + i * vr[(k - 1) + j * ldvr]
\]

for column major layout and

\[
v[k - 1) * ldvr + (j - 1)] + i * vr[(k - 1) * ldvr + j]
\]

for row major layout. Respectively, the k-th components of right eigenvector \( v_{j+1} \) can be computed as

\[
vr[(k - 1) + (j - 1) * ldvr] - i * vr[(k - 1) + j * ldvr]
\]

for column major layout and

\[
v[k - 1) * ldvr + (j - 1)] - i * vr[(k - 1) * ldvr + j]
\]

for row major layout.

For complex flavors:

The k-th component of the j-th right eigenvector \( v_j \) is stored in \( vr[(k - 1) + (j - 1) * ldvr] \) for column major layout and in \( vr[(k - 1) * ldvr + (j - 1)] \) for row major layout.

\[ilo, ihi\]

ilo and ihi are integer values such that on exit \( A_{i,j} = 0 \) and \( B_{i,j} = 0 \) if \( i > j \) and \( j = 1, \ldots, ilo-1 \) or \( i = ihi+1, \ldots, n \).

If \( \text{balanc} = 'N' \) or \( 'S' \), ilo = 1 and ihi = n.

\[lscale, rscale\]

Arrays, size at least \( \max(1, n) \) each.

\( lscale \) contains details of the permutations and scaling factors applied to the left side of \( A \) and \( B \).

If \( PL(j) \) is the index of the row interchanged with row \( j \), and \( DL(j) \) is the scaling factor applied to row \( j \), then

\[
lscale[j - 1] = PL(j), \text{ for } j = 1, \ldots, ilo-1
\]

\[
= DL(j), \text{ for } j = ilo, \ldots, ihi
\]

\[
= PL(j) \text{ for } j = ihi+1, \ldots, n.
\]

The order in which the interchanges are made is \( n \) to \( ihi+1 \), then \( 1 \) to \( ilo-1 \).

\( rscale \) contains details of the permutations and scaling factors applied to the right side of \( A \) and \( B \).

If \( PR(j) \) is the index of the column interchanged with column \( j \), and \( DR(j) \) is the scaling factor applied to column \( j \), then

\[
rscale[j - 1] = PR(j), \text{ for } j = 1, \ldots, ilo-1
\]

\[
= DR(j), \text{ for } j = ilo, \ldots, ihi
\]

\[
= PR(j) \text{ for } j = ihi+1, \ldots, n.
\]

The order in which the interchanges are made is \( n \) to \( ihi+1 \), then \( 1 \) to \( ilo-1 \).

\[abnrm, bbnrm\]

The one-norms of the balanced matrices \( A \) and \( B \), respectively.

\[rconde, rcondv\]

Arrays, size at least \( \max(1, n) \) each.

If \( \text{sense} = 'E' \), or \( 'B' \), \( rconde \) contains the reciprocal condition numbers of the eigenvalues, stored in consecutive elements of the array. For a complex conjugate pair of eigenvalues two consecutive elements of \( rconde \)
are set to the same value. Thus $rconde[j]$, $rcondv[j]$, and the $j$-th columns of $vl$ and $vr$ all correspond to the same eigenpair (but not in general the $j$-th eigenpair, unless all eigenpairs are selected).

If $sense = 'N'$, or 'V', $rconde$ is not referenced.

If $sense = 'V'$, or 'B', $rcondv$ contains the estimated reciprocal condition numbers of the eigenvectors, stored in consecutive elements of the array. For a complex eigenvector two consecutive elements of $rcondv$ are set to the same value.

If the eigenvalues cannot be reordered to compute , $rcondv[j]$ is set to 0; this can only occur when the true value would be very small anyway.

If $sense = 'N'$, or 'E', $rcondv$ is not referenced.

**Return Values**

This function returns a value $info$.

If $info=0$, the execution is successful.

If $info = -i$, the $i$-th parameter had an illegal value.

If $info = i$, and

i≤n: the QZ iteration failed. No eigenvectors have been calculated, but $alphar[j]$, $alphai[j]$ (for real flavors), or $alpha[j]$ (for complex flavors), and $beta[j]$, $j=info,\ldots, n - 1$ should be correct.

i > n: errors that usually indicate LAPACK problems:

i = n+1: other than QZ iteration failed in hgeqz;

i = n+2: error return from tgevc.

**Application Notes**

The quotients $alphar[j]/beta[j]$ and $alphai[j]/beta[j]$ may easily over- or underflow, and $beta[j]$ may even be zero. Thus, you should avoid simply computing the ratio. However, $alphar$ and $alphai$ (for real flavors) or $alpha$ (for complex flavors) will be always less than and usually comparable with $\text{norm}(A)$ in magnitude, and $beta$ always less than and usually comparable with $\text{norm}(B)$.

**?ggev3**

Computes the generalized eigenvalues and the left and right generalized eigenvectors for a pair of matrices.

**Syntax**

```c
lapack_int LAPACKE_sggev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
float * a, lapack_int lda, float * b, lapack_int ldb, float * alphar, float * alphai,
float * beta, float * vl, lapack_int ldvl, float * vr, lapack_int ldvr);

lapack_int LAPACKE_dggev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
double * a, lapack_int lda, double * b, lapack_int ldb, double * alphar, double * alphai,
double * beta, double * vl, lapack_int ldvl, double * vr, lapack_int ldvr);

lapack_int LAPACKE_cggev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n,
lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb,
lapack_complex_float * alpha, lapack_complex_float * beta, lapack_complex_float * vl,
lapack_int ldvl, lapack_complex_float * vr, lapack_int ldvr);
```
lapack_int LAPACKE_zggev3 (int matrix_layout, char jobvl, char jobvr, lapack_int n, 
lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb, 
lapack_complex_double * alpha, lapack_complex_double * beta, lapack_complex_double * 
vl, lapack_int ldvl, lapack_complex_double * vr, lapack_int ldvr);

Include Files

• mkl.h

Description

For a pair of \( n \)-by-\( n \) real or complex nonsymmetric matrices \((A, B)\), \( \text{?ggev3} \) computes the generalized eigenvalues, and optionally, the left and right generalized eigenvectors.

A generalized eigenvalue for a pair of matrices \((A, B)\) is a scalar \( \lambda \) or a ratio \( \alpha/\beta = \lambda \), such that \( A - \lambda B \) is singular. It is usually represented as the pair \((\alpha, \beta)\), as there is a reasonable interpretation for \( \beta = 0 \), and even for both being zero.

For real flavors:

The right eigenvector \( v_j \) corresponding to the eigenvalue \( \lambda_j \) of \((A, B)\) satisfies

\[ A * v_j = \lambda_j * B * v_j. \]

The left eigenvector \( u_j \) corresponding to the eigenvalue \( \lambda_j \) of \((A, B)\) satisfies

\[ u_j^H * A = \lambda_j * u_j^H * B \]

where \( u_j^H \) is the conjugate-transpose of \( u_j \).

For complex flavors:

The right generalized eigenvector \( v_j \) corresponding to the generalized eigenvalue \( \lambda_j \) of \((A, B)\) satisfies

\[ A * v_j = \lambda_j * B * v_j. \]

The left generalized eigenvector \( u_j \) corresponding to the generalized eigenvalues \( \lambda_j \) of \((A, B)\) satisfies

\[ u_j^H * A = \lambda_j * u_j^H * B \]

where \( u_j^H \) is the conjugate-transpose of \( u_j \).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **jobvl**
  = 'N': do not compute the left generalized eigenvectors;
  = 'V': compute the left generalized eigenvectors.

- **jobvr**
  = 'N': do not compute the right generalized eigenvectors;
  = 'V': compute the right generalized eigenvectors.

- **n**
  The order of the matrices \( A, B, VL, \) and \( VR \).
  \( n \geq 0. \)

- **a**
  Array, size \((lda*n)\).
  On entry, the matrix \( A \) in the pair \((A, B)\).

- **lda**
  The leading dimension of \( a \).
  \( lda \geq \text{max}(1,n) \).
Array, size ($ldb \times n$).
On entry, the matrix $B$ in the pair $(A, B)$.

The leading dimension of $b$.
$ldb \geq \max(1, n)$.

The leading dimension of the matrix $VL$.
$ldvl \geq 1$, and if $jobvl = 'V'$, $ldvl \geq n$.

The leading dimension of the matrix $VR$.
$ldvr \geq 1$, and if $jobvr = 'V'$, $ldvr \geq n$.

### Output Parameters

- $a$
  - On exit, $a$ is overwritten.

- $b$
  - On exit, $b$ is overwritten.

- $alphar$
  - Array, size ($n$).

- $alphai$
  - Array, size ($n$).

- $alpha$
  - Array, size ($n$).

- $beta$
  - Array, size ($n$).

  For real flavors:
  - On exit, $(alphar[j] + alphai[j]*i)/beta[j], j=0,\ldots,n-1$, are the generalized eigenvalues. If $alphai[j - 1]$ is zero, then the $j$-th eigenvalue is real; if positive, then the $j$-th and $(j+1)$-st eigenvalues are a complex conjugate pair, with $alphai[j]$ negative.

  Note: the quotients $alphar[j - 1]/beta[j - 1]$ and $alphai[j - 1]/beta[j - 1]$ can easily over- or underflow, and $beta[j]$ might even be zero. Thus, you should avoid computing the ratio $alpha/beta$ by simply dividing $alpha$ by $beta$. However, $alphar$ and $alphai$ are always less than and usually comparable with $\text{norm}(A)$ in magnitude, and $beta$ is always less than and usually comparable with $\text{norm}(B)$.

  For complex flavors:
  - On exit, $alpha[j]/beta[j], j=0,\ldots,n-1$, are the generalized eigenvalues.

  Note: the quotients $alpha[j - 1]/beta[j - 1]$ may easily over- or underflow, and $beta[j]$ can even be zero. Thus, you should avoid computing the ratio $alpha/beta$ by simply dividing $alpha$ by $beta$. However, $alpha$ is always less than and usually comparable with $\text{norm}(A)$ in magnitude, and $beta$ is always less than and usually comparable with $\text{norm}(B)$.

- $vl$
  - Array, size ($ldvl \times n$).

  For real flavors:
If \textit{jobvl} = 'V', the left eigenvectors \( u_j \) are stored one after another in the columns of \( vl \), in the same order as their eigenvalues. If the \( j \)-th eigenvalue is real, then \( u_j = \) the \( j \)-th column of \( vl \). If the \( j \)-th and \((j + 1)\)-st eigenvalues form a complex conjugate pair, then the real part of \( u_j = \) the \( j \)-th column of \( vl \) and the imaginary part of \( v_j = \) the \((j + 1)\)-st column of \( vl \).

Each eigenvector is scaled so the largest component has \( \text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1 \).

Not referenced if \textit{jobvl} = 'N'.

For complex flavors:

If \textit{jobvl} = 'V', the left generalized eigenvectors \( u_j \) are stored one after another in the columns of \( vl \), in the same order as their eigenvalues.

Each eigenvector is scaled so the largest component has \( \text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1 \).

Not referenced if \textit{jobvl} = 'N'.

\textit{vr}

Array, size \((ldvr*n)\).

For real flavors:

If \textit{jobvr} = 'V', the right eigenvectors \( v_j \) are stored one after another in the columns of \( vr \), in the same order as their eigenvalues. If the \( j \)-th eigenvalue is real, then \( v_j = \) the \( j \)-th column of \( vr \). If the \( j \)-th and \((j + 1)\)-st eigenvalues form a complex conjugate pair, then the real part of \( v_j = \) the \( j \)-th column of \( vr \) and the imaginary part of \( v_j = \) the \((j + 1)\)-st column of \( vr \).

Each eigenvector is scaled so the largest component has \( \text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1 \).

Not referenced if \textit{jobvr} = 'N'.

For complex flavors:

If \textit{jobvr} = 'V', the right generalized eigenvectors \( v_j \) are stored one after another in the columns of \( vr \), in the same order as their eigenvalues. Each eigenvector is scaled so the largest component has \( \text{abs}(\text{real part}) + \text{abs}(\text{imag. part}) = 1 \).

Not referenced if \textit{jobvr} = 'N'.

\textbf{Return Values}

This function returns a value \textit{info}.

\( = 0 \): successful exit
\( < 0 \): if \textit{info} = \(-i\), the \( i \)-th argument had an illegal value.
\( =1,\ldots,n \):

for real flavors:

The QZ iteration failed. No eigenvectors have been calculated, but \textit{alphar}[,], \textit{alphar}[j] and \textit{beta}[j] should be correct for \( j=\textit{info},\ldots,n - 1 \).

for complex flavors:
The QZ iteration failed. No eigenvectors have been calculated, but \( \alpha[j] \) and \( \beta[j] \) should be correct for \( j = \text{info}, \ldots, n-1 \).

\[
> n:
\]
\[
= n + 1: \text{other than QZ iteration failed in } \text{?hgeqz},
\]
\[
= n + 2: \text{error return from } \text{?tgevc}.
\]

## LAPACK Auxiliary Routines

Routine naming conventions, mathematical notation, and matrix storage schemes used for LAPACK auxiliary routines are the same as for the driver and computational routines described in previous chapters.

### \text{?lacgv}

**Conjugates a complex vector.**

**Syntax**

```
lapack_int LAPACKE_clacgv (lapack_int n, lapack_complex_float* x, lapack_int incx);
lapack_int LAPACKE_zlacgv (lapack_int n, lapack_complex_double* x, lapack_int incx);
```

**Include Files**

- `mkl.h`

**Description**

The routine conjugates a complex vector \( x \) of length \( n \) and increment \( \text{incx} \) (see "Vector Arguments in BLAS" in Appendix B).

**Input Parameters**

- \( n \)  
  The length of the vector \( x \) \( (n \geq 0) \).
- \( x \)  
  Array, dimension \((1+(n-1)\times |\text{incx}|)\).  
  Contains the vector of length \( n \) to be conjugated.
- \( \text{incx} \)  
  The spacing between successive elements of \( x \).

**Output Parameters**

- \( x \)  
  On exit, overwritten with \( \text{conjg}(x) \).

### \text{?lacrm}

**Multiplies a complex matrix by a square real matrix.**

**Syntax**

```
call clacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
call zlacrm( m, n, a, lda, b, ldb, c, ldc, rwork )
```

**Include Files**

- `mkl.h`
Description

The routine performs a simple matrix-matrix multiplication of the form
\[ C = A \cdot B, \]
where \( A \) is \( m \)-by-\( n \) and complex, \( B \) is \( n \)-by-\( n \) and real, \( C \) is \( m \)-by-\( n \) and complex.

Input Parameters

- \( m \) - INTEGER. The number of rows of the matrix \( A \) and of the matrix \( C \) \((m \geq 0)\).
- \( n \) - INTEGER. The number of columns and rows of the matrix \( B \) and the number of columns of the matrix \( C \) \((n \geq 0)\).
- \( a \) - COMPLEX for \texttt{clacrm}
  DOUBLE COMPLEX for \texttt{zlacrm}
  Array, \texttt{DIMENSION(lda, n)}. Contains the \( m \)-by-\( n \) matrix \( A \).
- \( lda \) - INTEGER. The leading dimension of the array \( a \), \( lda \geq \text{max}(1, m) \).
- \( b \) - REAL for \texttt{clacrm}
  DOUBLE PRECISION for \texttt{zlacrm}
  Array, \texttt{DIMENSION(ldb, n)}. Contains the \( n \)-by-\( n \) matrix \( B \).
- \( ldb \) - INTEGER. The leading dimension of the array \( b \), \( ldb \geq \text{max}(1, n) \).
- \( ldc \) - INTEGER. The leading dimension of the output array \( c \), \( ldc \geq \text{max}(1, n) \).
- \( rwork \) - REAL for \texttt{clacrm}
  DOUBLE PRECISION for \texttt{zlacrm}
  Workspace array, \texttt{DIMENSION(2*m*n)}.

Output Parameters

- \( c \) - COMPLEX for \texttt{clacrm}
  DOUBLE COMPLEX for \texttt{zlacrm}
  Array, \texttt{DIMENSION(ldc, n)}. Contains the \( m \)-by-\( n \) matrix \( C \).

\texttt{?syconv}

Converts a symmetric matrix given by a triangular matrix factorization into two matrices and vice versa.

Syntax

\begin{verbatim}
lapack_int LAPACKE_ssyconv (int matrix_layout, char uplo, char way, lapack_int n, float * a, lapack_int lda, const lapack_int * ipiv, float * e);

lapack_int LAPACKE_dsyconv (int matrix_layout, char uplo, char way, lapack_int n, double* a, lapack_int lda, const lapack_int * ipiv, double * e);
\end{verbatim}
lapack_int LAPACKE_csyconv (int matrix_layout, char uplo, char way, lapack_int n,
lapack_complex_float * a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_float * e);

lapack_int LAPACKE_zsyconv (int matrix_layout, char uplo, char way, lapack_int n,
lapack_complex_double* a, lapack_int lda, const lapack_int * ipiv,
lapack_complex_double * e);

Include Files

• mkl.h

Description

The routine converts matrix $A$, which results from a triangular matrix factorization, into matrices $L$ and $D$ and vice versa. The routine returns non-diagonalized elements of $D$ and applies or reverses permutation done with the triangular matrix factorization.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Must be 'U' or 'L'.
  Indicates whether the details of the factorization are stored as an upper or lower triangular matrix:
  If $uplo = 'U'$: the upper triangular, $A = U*D*U^T$.
  If $uplo = 'L'$: the lower triangular, $A = L*D*L^T$.

- **way**
  Must be 'C' or 'R'.

- **n**
  The order of matrix $A$; $n \geq 0$.

- **a**
  Array of size max(1,lda *n).
  The block diagonal matrix $D$ and the multipliers used to obtain the factor $U$ or $L$ as computed by ?sytrf.

- **lda**
  The leading dimension of $a$; $lda \geq \text{max}(1, n)$.

- **ipiv**
  Array, size at least max(1, n).
  Details of the interchanges and the block structure of $D$, as returned by ?sytrf.

Output Parameters

- **e**
  Array of size max(1, n) containing the superdiagonal/subdiagonal of the symmetric 1-by-1 or 2-by-2 block diagonal matrix $D$ in $L*D*L^T$.

Return Values

- **info**
  If $info = 0$, the execution is successful.
  If $info < 0$, the $i$-th parameter had an illegal value.
  If $info = -1011$, memory allocation error occurred.
See Also

?sytrf

?syr
Performs the symmetric rank-1 update of a complex symmetric matrix.

Syntax

```c
lapack_int LAPACKE_csyr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float alpha, const lapack_complex_float * x, lapack_int incx,
lapack_complex_float * a, lapack_int lda);
```

```c
lapack_int LAPACKE_zsyr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double alpha, const lapack_complex_double * x, lapack_int incx,
lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine performs the symmetric rank 1 operation defined as

\[ a := \alpha x^H x + a, \]

where:
- \( \alpha \) is a complex scalar.
- \( x \) is an \( n \)-element complex vector.
- \( a \) is an \( n \times n \) complex symmetric matrix.

These routines have their real equivalents in BLAS (see ?syr in Chapter 2).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **uplo**
  Specifies whether the upper or lower triangular part of the array \( a \) is used:
  - If \( \text{uplo} = \text{'U'} \) or \( \text{'u'} \), then the upper triangular part of the array \( a \) is used.
  - If \( \text{uplo} = \text{'L'} \) or \( \text{'l'} \), then the lower triangular part of the array \( a \) is used.

- **n**
  Specifies the order of the matrix \( a \). The value of \( n \) must be at least zero.

- **alpha**
  Specifies the scalar \( \alpha \).

- **x**
  Array, size at least \((1 + (n - 1) \times \text{abs}(\text{incx}))\). Before entry, the incremented array \( x \) must contain the \( n \)-element vector \( x \).

- **incx**
  Specifies the increment for the elements of \( x \). The value of \( \text{incx} \) must not be zero.

- **a**
  Array, size max(1, \( \text{lda} \times n \)). Before entry with \( \text{uplo} = \text{'U'} \) or \( \text{'u'} \), the leading \( n \)-by-\( n \) upper triangular part of the array \( a \) must contain the upper triangular part of the symmetric matrix and the strictly lower triangular part of \( a \) is not referenced.
Before entry with `uplo = 'L'` or `l'`, the leading \(n\)-by-\(n\) lower triangular part of the array \(a\) must contain the lower triangular part of the symmetric matrix and the strictly upper triangular part of \(a\) is not referenced.

\textbf{lda}

Specifies the leading dimension of \(a\) as declared in the calling (sub)program. The value of \(lda\) must be at least \(\max(1, n)\).

\textbf{Output Parameters}

\textbf{a}

With `uplo = 'U'` or `u'`, the upper triangular part of the array \(a\) is overwritten by the upper triangular part of the updated matrix.

With `uplo = 'L'` or `l'`, the lower triangular part of the array \(a\) is overwritten by the lower triangular part of the updated matrix.

\textbf{Return Values}

This function returns a value \(info\).

If \(info = 0\), the execution is successful.

If \(info < 0\), the \(i\)-th parameter had an illegal value.

If \(info = -1011\), memory allocation error occurred.

\textbf{i?max1}

\textit{Finds the index of the vector element whose real part has maximum absolute value.}

\textbf{Syntax}

\begin{verbatim}
MKL_INT icmax1(const MKL_INT *n, const MKL_Complex8 *cx, const MKL_INT *incx)
MKL_INT izmax1(const MKL_INT *n, const MKL_Complex16 *cx, const MKL_INT *incx)
\end{verbatim}

\textbf{Include Files}

\begin{itemize}
  \item mkl.h
\end{itemize}

\textbf{Description}

Given a complex vector \(cx\), the \textbf{i?max1} functions return the index of the first vector element of maximum absolute value. These functions are based on the BLAS functions \textit{icamax}/\textit{izamax}, but using the absolute value of components. They are designed for use with \textit{clacon}/\textit{zlacon}.

\textbf{Input Parameters}

\begin{itemize}
  \item \textit{n}
    \begin{itemize}
      \item Specifies the number of elements in the vector \(cx\).
    \end{itemize}
  \item \textit{cx}
    \begin{itemize}
      \item \textit{Array, size at least} \((1+(n-1)\text{abs}(\text{incx}))\).
      \item Contains the input vector.
    \end{itemize}
  \item \textit{incx}
    \begin{itemize}
      \item Specifies the spacing between successive elements of \(cx\).
    \end{itemize}
\end{itemize}

\textbf{Return Values}

Index of the vector element of maximum absolute value.
**scsum1**
*Forms the 1-norm of the complex vector using the true absolute value.*

**Syntax**

```c
float scsum1(const MKL_INT*n, const MKL_Complex8*cx, const MKL_INT*incx)
double dzsum1(const MKL_INT*n, const MKL_Complex16*cx, const MKL_INT*incx)
```

**Include Files**

- mkl.h

**Description**

Given a complex vector \( cx \), \( scsum1/dzsum1 \) functions take the sum of the absolute values of vector elements and return a single/double precision result, respectively. These functions are based on \( scasum/dzasum \) from Level 1 BLAS, but use the true absolute value and were designed for use with \( clacon/zlacon \).

**Input Parameters**

- \( n \)  
  Specifies the number of elements in the vector \( cx \).

- \( cx \)  
  Array, size at least \( (1+(n-1)\times\text{abs}(incx)) \).  
  Contains the input vector whose elements will be summed.

- \( incx \)  
  Specifies the spacing between successive elements of \( cx \) (\( incx > 0 \)).

**Return Values**

Sum of absolute values.

**zgelq2**
*Computes the LQ factorization of a general rectangular matrix using an unblocked algorithm.*

**Syntax**

```c
lapack_int LAPACKE_sgelq2 (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);
lapack_int LAPACKE_dgelq2 (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);
lapack_int LAPACKE_cgelq2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);
lapack_int LAPACKE_zgelq2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);
```

**Include Files**

- mkl.h

**Description**

The routine computes an \( LQ \) factorization of a real/complex \( m \)-by-\( n \) matrix \( A \) as \( A = LQ \).
The routine does not form the matrix \( Q \) explicitly. Instead, \( Q \) is represented as a product of \( \min(m, n) \) elementary reflectors:

\[
Q = H(k) \ldots H(2) H(1) \quad (\text{or } Q = H(k)H^\dagger \ldots H(2)H(1)H^\dagger \text{ for complex flavors}), \quad \text{where } k = \min(m, n)
\]

Each \( H(i) \) has the form

\[
H(i) = I - \tau v^* v^T \quad \text{for real flavors, or}
\]

\[
H(i) = I - \tau v^* H^\dagger \quad \text{for complex flavors},
\]

where \( \tau \) is a real/complex scalar stored in \( \tau(i) \), and \( v \) is a real/complex vector with \( v_{1:i-1} = 0 \) and \( v_i = 1 \).

On exit, the \( j \)-th \( (i+1 \leq j \leq n) \) component of vector \( v \) (for real functions) or its conjugate (for complex functions) is stored in \( a[i - 1 + \text{lda}^*(j - 1)] \) for column major layout or in \( a[j - 1 + \text{lda}^*(i - 1)] \) for row major layout.

**Input Parameters**

- \( m \)  
  The number of rows in the matrix \( A \) \( (m \geq 0) \).

- \( n \)  
  The number of columns in \( A \) \( (n \geq 0) \).

- \( a \)  
  Array, size at least \( \max(1, \text{lda}^*n) \) for column major and \( \max(1, \text{lda}^*m) \) for row major layout. Array \( a \) contains the \( m \)-by-\( n \) matrix \( A \).

- \( \text{lda} \)  
  The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

**Output Parameters**

- \( a \)  
  Overwritten by the factorization data as follows:

  on exit, the elements on and below the diagonal of the array \( a \) contain the \( m \)-by-\( \min(n,m) \) lower trapezoidal matrix \( L \) \( (L \) is lower triangular if \( n \geq m) \); the elements above the diagonal, with the array \( \tau \), represent the orthogonal/unitary matrix \( Q \) as a product of \( \min(n,m) \) elementary reflectors.

- \( \tau \)  
  Array, size at least \( \max(1, \min(m, n)) \).

  Contains scalar factors of the elementary reflectors.

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = -1011 \), memory allocation error occurred.

**?geqr2**

Computes the QR factorization of a general rectangular matrix using an unblocked algorithm.

**Syntax**

```c
lapack_int LAPACKE_sgeqr2 (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, float* tau);
```
lapack_int LAPACKE_dgeqr2 (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, double* tau);

lapack_int LAPACKE_cgeqr2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_complex_float* tau);

lapack_int LAPACKE_zgeqr2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_complex_double* tau);

**Include Files**
- mkl.h

**Description**
The routine computes a QR factorization of a real/complex $m$-by-$n$ matrix $A$ as $A = Q \cdot R$.
The routine does not form the matrix $Q$ explicitly. Instead, $Q$ is represented as a product of $\min(m, n)$ 
**elementary reflectors** :

$$Q = H(1) \cdot H(2) \cdots \cdot H(k), \text{ where } k = \min(m, n).$$

Each $H(i)$ has the form

$$H(i) = I - \tau * v * v^T$$
for real flavors, or

$$H(i) = I - \tau * v * v^H$$
for complex flavors

where $\tau$ is a real/complex scalar stored in $\tau[i]$, and $v$ is a real/complex vector with $v_{1:i-1} = 0$ and $v_i = 1$.

On exit, $v_{i+1:m}$ is stored in $a(i+1:m, i)$.

**Input Parameters**
- $m$ The number of rows in the matrix $A$ ($m \geq 0$).
- $n$ The number of columns in $A$ ($n \geq 0$).
- $a$ Array, size at least $\max(1, lda \cdot n)$ for column major and $\max(1, lda \cdot m)$ for row major layout. Array $a$ contains the $m$-by-$n$ matrix $A$.
- $lda$ The leading dimension of $a$; at least $\max(1, m)$ for column major layout and $\max(1, n)$ for row major layout.

**Output Parameters**
- $a$ Overwritten by the factorization data as follows:
  on exit, the elements on and above the diagonal of the array $a$ contain the $\min(n,m)$-by-$n$ upper trapezoidal matrix $R$ ($R$ is upper triangular if $n \geq n$); the elements below the diagonal, with the array $\tau$, represent the orthogonal/unitary matrix $Q$ as a product of elementary reflectors.
- $\tau$ Array, size at least $\max(1, \min(m, n))$.
  Contains scalar factors of the elementary reflectors.

**Return Values**
This function returns a value $info$.
If $info = 0$, the execution is successful.
If $info = -i$, the $i$-th parameter had an illegal value.
If $info = -1011$, memory allocation error occurred.

**?geqrt2**

Computes a QR factorization of a general real or complex matrix using the compact WY representation of $Q$.

**Syntax**

```c
lapack_int LAPACKE_sgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, float * a, lapack_int lda, float * t, lapack_int ldt );
```

```c
lapack_int LAPACKE_dgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, double * a, lapack_int lda, double * t, lapack_int ldt );
```

```c
lapack_int LAPACKE_cgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_complex_float * t, lapack_int ldt );
```

```c
lapack_int LAPACKE_zgeqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * t, lapack_int ldt );
```

**Include Files**
- mkl.h

**Description**

The strictly lower triangular matrix $V$ contains the elementary reflectors $H(i)$ in the $i$th column below the diagonal. For example, if $m=5$ and $n=3$, the matrix $V$ is

$$
V = \begin{bmatrix}
1 \\
v_1 & 1 \\
v_1 & v_2 & 1 \\
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3 \\
\end{bmatrix}
$$

where $v_i$ represents the vector that defines $H(i)$. The vectors are returned in the lower triangular part of array $a$.

**NOTE**
The 1s along the diagonal of $V$ are not stored in $a$.

The block reflector $H$ is then given by

$H = I - V^T V$ for real flavors, and

$H = I - V^* V^H$ for complex flavors,

where $V^T$ is the transpose and $V^H$ is the conjugate transpose of $V$. 
Input Parameters

\( m \)

The number of rows in the matrix \( A \) \((m \geq n)\).

\( n \)

The number of columns in \( A \) \((n \geq 0)\).

\( a \)

Array, size at least \( \max(1, \text{lda} \cdot n) \) for column major and \( \max(1, \text{lda} \cdot m) \) for row major layout. Array \( a \) contains the \( m \)-by-\( n \) matrix \( A \).

\( \text{lda} \)

The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1,n) \) for row major layout.

\( \text{ldt} \)

The leading dimension of \( t \); at least \( \max(1, n) \).

Output Parameters

\( a \)

Overwritten by the factorization data as follows:

The elements on and above the diagonal of the array contain the \( n \)-by-\( n \) upper triangular matrix \( R \). The elements below the diagonal are the columns of \( V \).

\( t \)

Array, size at least \( \max(1, \text{ldt} \cdot n) \).

The \( n \)-by-\( n \) upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector \( T \). The elements below the diagonal are not used.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info < 0 \) and \( info = -i \), the \( i \)th argument had an illegal value.

If \( info = -1011 \), memory allocation error occurred.

\(?\text{geqrt3}\)

Recursively computes a QR factorization of a general real or complex matrix using the compact WY representation of \( Q \).

Syntax

\[
\text{lapack_int LAPACKE_sgeqrt3 (int matrix_layout, lapack_int m, lapack_int n, float * a, lapack_int lda, float * t, lapack_int ldt \});
\]

\[
\text{lapack_int LAPACKE_dgeqrt3 (int matrix_layout, lapack_int m, lapack_int n, double * a, lapack_int lda, double * t, lapack_int ldt \});
\]

\[
\text{lapack_int LAPACKE_cgeqrt3 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_complex_float * t, lapack_int ldt \});
\]

\[
\text{lapack_int LAPACKE_zgeqrt3 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * t, lapack_int ldt \});
\]
Include Files

- mkl.h

Description

The strictly lower triangular matrix \( V \) contains the elementary reflectors \( H(i) \) in the \( i \)th column below the diagonal. For example, if \( m=5 \) and \( n=3 \), the matrix \( V \) is

\[
V = \begin{bmatrix}
1 & & \\
v_1 & 1 & \\
v_1 & v_2 & 1 \\
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3
\end{bmatrix}
\]

where \( v_i \) represents one of the vectors that define \( H(i) \). The vectors are returned in the lower part of the triangular array \( a \).

**NOTE**
The 1s along the diagonal of \( V \) are not stored in \( a \).

The block reflector \( H \) is then given by

\[
H = I - V^* V^T \quad \text{for real flavors, and}
\]

\[
H = I - V^* V^H \quad \text{for complex flavors,}
\]

where \( V^T \) is the transpose and \( V^H \) is the conjugate transpose of \( V \).

Input Parameters

- \( m \)
  The number of rows in the matrix \( A (m \geq n) \).

- \( n \)
  The number of columns in \( A (n \geq 0) \).

- \( a \)
  Array, size at least \( \max(1, lda \times n) \) for column major and \( \max(1, lda \times m) \) for row major layout. Array \( a \) contains the \( m \)-by-\( n \) matrix \( A \).

- \( lda \)
  The leading dimension of \( a \); at least \( \max(1, m) \) for column major layout and \( \max(1,n) \) for row major layout.

- \( ldt \)
  The leading dimension of \( t \); at least \( \max(1, n) \).

Output Parameters

- \( a \)
  The elements on and above the diagonal of the array contain the \( n \)-by-\( n \) upper triangular matrix \( R \). The elements below the diagonal are the columns of \( V \).

- \( t \)
  Array, size \( ldt \) by \( n \).
The $n$-by-$n$ upper triangular factor of the block reflector. The elements on and above the diagonal contain the block reflector $T$. The elements below the diagonal are not used.

**Return Values**

This function returns a value $info$.

- If $info = 0$, the execution is successful.
- If $info < 0$ and $info = -i$, the $i$th argument had an illegal value.
- If $info = -1011$, memory allocation error occurred.

**?getf2**

*Computes the LU factorization of a general $m$-by-$n$ matrix using partial pivoting with row interchanges (unblocked algorithm).*

**Syntax**

```c
lapack_int LAPACKE_sgetf2 (int matrix_layout, lapack_int m, lapack_int n, float* a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_dgetf2 (int matrix_layout, lapack_int m, lapack_int n, double* a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_cgetf2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_float* a, lapack_int lda, lapack_int * ipiv);

lapack_int LAPACKE_zgetf2 (int matrix_layout, lapack_int m, lapack_int n, lapack_complex_double* a, lapack_int lda, lapack_int * ipiv);
```

**Include Files**

- mkl.h

**Description**

The routine computes the $LU$ factorization of a general $m$-by-$n$ matrix $A$ using partial pivoting with row interchanges. The factorization has the form

$$A = P*L*U$$

where $P$ is a permutation matrix, $L$ is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and $U$ is upper triangular (upper trapezoidal if $m < n$).

**Input Parameters**

- $m$
  - The number of rows in the matrix $A$ ($m \geq 0$).

- $n$
  - The number of columns in $A$ ($n \geq 0$).

- $a$
  - Array, size at least $\max(1, lda*n)$ for column major and $\max(1, lda*m)$ for row major layout. Array $a$ contains the $m$-by-$n$ matrix $A$.

- $lda$
  - The leading dimension of $a$; at least $\max(1, m)$ for column major layout and $\max(1,n)$ for row major layout.
Output Parameters

\( a \)  
Overwritten by \( L \) and \( U \). The unit diagonal elements of \( L \) are not stored.

\( ipiv \)  
Array, size at least \( \max(1, \min(m,n)) \).
The pivot indices: for \( 1 \leq i \leq n \), row \( i \) was interchanged with row \( ipiv(i) \).

Return Values

This function returns a value \( info \).

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = i > 0 \), \( u_{ii} \) is 0. The factorization has been completed, but \( U \) is exactly singular. Division by 0 will occur if you use the factor \( U \) for solving a system of linear equations.

If \( info = -1011 \), memory allocation error occurred.

\texttt{lacn2}

Estimates the 1-norm of a square matrix, using reverse communication for evaluating matrix-vector products.

Syntax

C:

\begin{verbatim}
lapack_int LAPACKE_slacn2 (lapack_int n, float * v, float * x, lapack_int * isgn, float * est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_clacn2 (lapack_int n, lapack_complex_float * v, lapack_complex_float * x, float * est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_dlacn2 (lapack_int n, double * v, double * x, lapack_int * isgn, double * est, lapack_int * kase, lapack_int * isave);

lapack_int LAPACKE_zlacn2 (lapack_int n, lapack_complex_double * v, lapack_complex_double * x, double * est, lapack_int * kase, lapack_int * isave);
\end{verbatim}

Include Files

- mkl.h

Description

The routine estimates the 1-norm of a square, real or complex matrix \( A \). Reverse communication is used for evaluating matrix-vector products.

Input Parameters

\( n \)  
The order of the matrix \( A \) \((n \geq 1)\).

\( v, x \)  
Arrays, size \((n)\) each.

\( v \) is a workspace array.

\( x \) is used as input after an intermediate return.

\( isgn \)  
Workspace array, size \((n)\), used with real flavors only.
On entry with kase set to 1 or 2, and isave(1) = 1, est must be unchanged from the previous call to the routine.

On the initial call to the routine, kase must be set to 0.

Array, size (3).
Contains variables from the previous call to the routine.

Output Parameters

An estimate (a lower bound) for norm(A).

On an intermediate return, kase is set to 1 or 2, indicating whether x is overwritten by A*x or A^T*x for real flavors and A*x or A^H*x for complex flavors.

On the final return, kase is set to 0.

On the final return, v = A*w, where est = norm(v)/norm(w) (w is not returned).

On an intermediate return, x is overwritten by
A*x, if kase = 1,
A^T*x, if kase = 2 (for real flavors),
A^H*x, if kase = 2 (for complex flavors),
and the routine must be re-called with all the other parameters unchanged.

This parameter is used to save variables between calls to the routine.

Return Values

This function returns a value info.
If info = 0, the execution is successful.
If info = -i, the i-th parameter had an illegal value.

?lacpy
Copies all or part of one two-dimensional array to another.

Syntax

lapack_int LAPACKE_slacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n, const float* a, lapack_int lda, float* b, lapack_int ldb);
lapack_int LAPACKE_dlacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n, const double* a, lapack_int lda, double* b, lapack_int ldb);
lapack_int LAPACKE_clacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n, const lapack_complex_float* a, lapack_int lda, lapack_complex_float* b, lapack_int ldb);
lapack_int LAPACKE_zlacpy (int matrix_layout, char uplo, lapack_int m, lapack_int n, const lapack_complex_double* a, lapack_int lda, lapack_complex_double* b, lapack_int ldb);
Include Files

- mkl.h

Description

The routine copies all or part of a two-dimensional matrix \( A \) to another matrix \( B \).

Input Parameters

\( uplo \)

Specifies the part of the matrix \( A \) to be copied to \( B \).

If \( uplo = 'U' \), the upper triangular part of \( A \);
if \( uplo = 'L' \), the lower triangular part of \( A \).
Otherwise, all of the matrix \( A \) is copied.

\( m \)

The number of rows in the matrix \( A \) \((m \geq 0)\).

\( n \)

The number of columns in \( A \) \((n \geq 0)\).

\( a \)

Array, size at least \( \max(1, lda*n) \) for column major and \( \max(1, lda*m) \) for row major layout. \( A \) contains the \( m \)-by-\( n \) matrix \( A \).
If \( uplo = 'U' \), only the upper triangle or trapezoid is accessed; if \( uplo = 'L' \), only the lower triangle or trapezoid is accessed.

\( lda \)

The leading dimension of \( a \); \( lda \geq \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

\( ldb \)

The leading dimension of the output array \( b \); \( ldb \geq \max(1, m) \) for column major layout and \( \max(1, n) \) for row major layout.

Output Parameters

\( b \)

Array, size at least \( \max(1, ldb*n) \) for column major and \( \max(1, ldb*m) \) for row major layout. Array \( a \) contains the \( m \)-by-\( n \) matrix \( B \).
On exit, \( B = A \) in the locations specified by \( uplo \).

Return Values

This function returns a value \( info \).
If \( info = 0 \), the execution is successful.
If \( info = -i \), the \( i \)-th parameter had an illegal value.

?lakf2

Forms a matrix containing Kronecker products between the given matrices.

Syntax

\[
\text{void slakf2 (lapack_int *m, lapack_int *n, float *a, lapack_int *lda, float *b, float *d, float *e, float *z, lapack_int *ldz);}\\
\text{void dlakf2 (lapack_int *m, lapack_int *n, double *a, lapack_int *lda, double *b, double *d, double *e, double *z, lapack_int *ldz);}\\
\]
void clakf2 (lapack_int *m, lapack_int *n, lapack_complex *a, lapack_int *lda, 
      lapack_complex *b, lapack_complex *d, lapack_complex *e, lapack_complex *z, lapack_int *ldz);
void zlakf2 (lapack_int *m, lapack_int *n, lapack_complex_double *a, lapack_int *lda, 
      lapack_complex_double *b, lapack_complex_double *d, lapack_complex_double *e, 
      lapack_complex_double *z, lapack_int *ldz);

Include Files
• mkl.h

Description
The routine ?lakf2 forms the 2*m*n by 2*m*n matrix Z.

\[ Z = \begin{bmatrix}
    \text{kron}(I_n, A) & -\text{kron}(B^T, I_m) \\
    \text{kron}(I_n, D) & -\text{kron}(B^T, I_m)
\end{bmatrix}, \]

where In is the identity matrix of size n and X^T is the transpose of X. kron(X, Y) is the Kronecker product between the matrices X and Y.

Input Parameters

- \( m \) Size of matrix, \( m \geq 1 \)
- \( n \) Size of matrix, \( n \geq 1 \)
- \( a \) Array, size lda-by-n. The matrix A in the output matrix Z.
- \( lda \) The leading dimension of a, b, d, and e. \( lda \geq m+n \).
- \( b \) Array, size lda by n. Matrix used in forming the output matrix Z.
- \( d \) Array, size lda by m. Matrix used in forming the output matrix Z.
- \( e \) Array, size lda by n. Matrix used in forming the output matrix Z.
- \( ldz \) The leading dimension of Z. \( ldz \geq 2 \times m \times n \).

Output Parameters

- \( z \) Array, size ldz-by-2*m*n. The resultant Kronecker m*n*2-by-m*n*2 matrix.

?lange

Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a general rectangular matrix.

Syntax

float LAPACKE_slange (int matrix_layout, char norm, lapack_int m, lapack_int n, const float * a, lapack_int lda);
double LAPACKE_dlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const double * a, lapack_int lda);
float LAPACKE_clange (int matrix_layout, char norm, lapack_int m, lapack_int n, const lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlange (int matrix_layout, char norm, lapack_int m, lapack_int n, const lapack_complex_double * a, lapack_int lda);

Include Files
• mkl.h

Description
The function ?lange returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex matrix A.

Input Parameters

specifies the value to be returned by the routine:
= 'M' or 'm': val = max(abs(Aij)), largest absolute value of the matrix A.
= '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': val = normI(A), infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': val = normF(A), Frobenius norm of the matrix A (square root of sum of squares).

m
The number of rows of the matrix A.
m≥ 0. When m = 0, ?lange is set to zero.

n
The number of columns of the matrix A.
n≥ 0. When n = 0, ?lange is set to zero.

Array, size at least max(1, lda*n) for column major and max(1, lda*m) for row major layout. Array a contains the m-by-n matrix A.

lda
The leading dimension of the array a.
lda≥ max(n,1) for column major layout and max(1,n) for row major layout.

?lansy
Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a real/complex symmetric matrix.

Syntax
float LAPACKE_slansy (int matrix_layout, char norm, char uplo, lapack_int n, const float * a, lapack_int lda);

double LAPACKE_dlansy (int matrix_layout, char norm, char uplo, lapack_int n, const double * a, lapack_int lda);
float LAPACKE_clansy (int matrix_layout, char norm, char uplo, lapack_int n, const
       lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlansy (int matrix_layout, char norm, char uplo, lapack_int n, const
       lapack_complex_double * a, lapack_int lda);

Include Files

- mkl.h

Description

The function ?lansy returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the
element of largest absolute value of a real/complex symmetric matrix A.

Input Parameters

- **norm**
  Specifies the value to be returned by the routine:
  = 'M' or 'm': val = max(abs(A_{ij})), largest absolute value of the matrix A.
  = '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A (maximum column sum),
  = 'I' or 'i': val = normI(A), infinity norm of the matrix A (maximum row sum),
  = 'F', 'f', 'E' or 'e': val = normF(A), Frobenius norm of the matrix A (square root of sum of squares).

- **uplo**
  Specifies whether the upper or lower triangular part of the symmetric matrix A is to be referenced.
  = 'U': Upper triangular part of A is referenced.
  = 'L': Lower triangular part of A is referenced

- **n**
  The order of the matrix A. \( n \geq 0 \). When \( n = 0 \), ?lansy is set to zero.

- **a**
  Array, size at least max(1,lda*n). The symmetric matrix A.
  If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of a contains the
  upper triangular part of the matrix A, and the strictly lower triangular part
  of a is not referenced.
  If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of a contains the
  lower triangular part of the matrix A, and the strictly upper triangular part
  of a is not referenced.

- **lda**
  The leading dimension of the array a.
  \( lda \geq \max(n,1) \).

?lanhe

*Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix.*
Syntax

float LAPACKE_clanhe (int matrix_layout, char norm, char uplo, lapack_int n, const lapack_complex_float * a, lapack_int lda);

double LAPACKE_zlanhe (int matrix_layout, char norm, char uplo, lapack_int n, const lapack_complex_double * a, lapack_int lda);

Include Files
• mkl.h

Description
The function ?lanhe returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a complex Hermitian matrix A.

Input Parameters

\( \text{norm} \)

Specifies the value to be returned by the routine:

\( = 'M' \) or \( 'm' \): \( val = \max(\text{abs}(A)) \), largest absolute value of the matrix A.
\( = '1' \) or \( 'O' \) or \( 'o' \): \( val = \text{norm1}(A) \), 1-norm of the matrix A (maximum column sum),
\( = 'I' \) or \( 'i' \): \( val = \text{normI}(A) \), infinity norm of the matrix A (maximum row sum),
\( = 'F', 'f', 'E' \) or \( 'e' \): \( val = \text{normF}(A) \), Frobenius norm of the matrix A (square root of sum of squares).

\( \text{uplo} \)

Specifies whether the upper or lower triangular part of the Hermitian matrix A is to be referenced.

\( = 'U' \): Upper triangular part of A is referenced.
\( = 'L' \): Lower triangular part of A is referenced

\( n \)

The order of the matrix A. \( n \geq 0 \). When \( n = 0 \), ?lanhe is set to zero.

\( a \)

Array, size at least \( \max(1, lda*n) \). The Hermitian matrix A.

If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( a \) contains the upper triangular part of the matrix \( A \), and the strictly lower triangular part of \( a \) is not referenced.

If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( a \) contains the lower triangular part of the matrix \( A \), and the strictly upper triangular part of \( a \) is not referenced.

\( lda \)

The leading dimension of the array \( a \).

\( lda \geq \max(n,1) \).

?lantr

Returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix.
Syntax

float LAPACKE_slantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const float * a, lapack_int * lda, float * work);
double LAPACKE_dlantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const double * a, lapack_int * lda, double * work);
float LAPACKE_clantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const lapack_complex_float * a, lapack_int * lda, float * work);
double LAPACKE_zlantr (char * norm, char * uplo, char * diag, lapack_int * m, lapack_int * n, const lapack_complex_double * a, lapack_int * lda, double * work);

Include Files

• mkl.h

Description

The function ?lantr returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular matrix A.

Input Parameters

norm

Specifies the value to be returned by the routine:
= 'M' or 'm': \( \text{val} = \max(\text{abs}(A_{ij})) \), largest absolute value of the matrix A.
= '1' or '0' or 'o': \( \text{val} = \text{norm}_1(A) \), 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': \( \text{val} = \text{norm}_\infty(A) \), infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': \( \text{val} = \text{norm}_F(A) \), Frobenius norm of the matrix A (square root of sum of squares).

uplo

Specifies whether the matrix A is upper or lower trapezoidal.
= 'U': Upper trapezoidal
= 'L': Lower trapezoidal.

Note that A is triangular instead of trapezoidal if \( m = n \).

diag

Specifies whether or not the matrix A has unit diagonal.
= 'N': Non-unit diagonal
= 'U': Unit diagonal.

m

The number of rows of the matrix A. \( m \geq 0 \), and if uplo = 'U', \( m \leq n \).

When \( m = 0 \), ?lantr is set to zero.

n

The number of columns of the matrix A. \( n \geq 0 \), and if uplo = 'L', \( n \leq m \).

When \( n = 0 \), ?lantr is set to zero.

a

Array, size at least \( \max(1, \text{lda} \times n) \) for column major and \( \max(1, \text{lda} \times m) \) for row major layout.
The trapezoidal matrix $A$ ($A$ is triangular if $m = n$).

If $\text{uplo} = 'U'$, the leading $m$-by-$n$ upper trapezoidal part of the array $a$ contains the upper trapezoidal matrix, and the strictly lower triangular part of $A$ is not referenced.

If $\text{uplo} = 'L'$, the leading $m$-by-$n$ lower trapezoidal part of the array $a$ contains the lower trapezoidal matrix, and the strictly upper triangular part of $A$ is not referenced. Note that when $\text{diag} = 'U'$, the diagonal elements of $A$ are not referenced and are assumed to be one.

**lda**

The leading dimension of the array $a$.

$lda \geq \max(m,1)$ for column major layout and $\geq \max(1,n)$ for row major layout.

---

**LAPACKE_set_nancheck**

*Turns NaN checking off or on*

LAPACKE_set_nancheck(int flag);

**Description**

The routine sets a value for the LAPACKE NaN checking flag, which indicates whether or not LAPACKE routines check input matrices for NaNs.

**Input Parameters**

*flag*;

If $flag = 0$, NaN checking is turned OFF. Otherwise, it is turned ON.

**LAPACKE_get_nancheck**

*Gets the current NaN checking flag, which indicates whether NaN checking has been turned off or on.*

int flag = LAPACKE_get_nancheck();

**Description**

The function returns the current value for the LAPACKE NaN checking flag, which indicates whether or not LAPACKE routines check input matrices for NaNs.

**Return Value**

An integer value is returned which indicates the current NaN checking status.

The returned flag value is either 0 (OFF) or 1 (ON), even though any integer value can be used as an input parameter for LAPACKE_set_nancheck.

For example, the following code turns on NaN checking:

LAPACKE_set_nancheck(100);
int flag = LAPACKE_get_nancheck(); // flag==1, not 100.

---

**?lapmr**

*Rearranges rows of a matrix as specified by a permutation vector.*
Syntax

```c
lapack_int LAPACKE_slapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, float* x, lapack_int ldx, lapack_int * k);
```

```c
lapack_int LAPACKE_dlapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, double* x, lapack_int ldx, lapack_int * k);
```

```c
lapack_int LAPACKE_clapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, lapack_complex_float* x, lapack_int ldx, lapack_int * k);
```

```c
lapack_int LAPACKE_zlapmr (int matrix_layout, lapack_logical forwrd, lapack_int m,
lapack_int n, lapack_complex_double* x, lapack_int ldx, lapack_int * k);
```

Include Files

- mkl.h

Description

The ?lapmr routine rearranges the rows of the \( m \)-by-\( n \) matrix \( X \) as specified by the permutation \( k[0], k[1], \ldots, k[m-1] \) of the integers \( 1, \ldots, m \).

If \( \text{forwrd} \) is true, forward permutation:

\[ X(k[i-1,:]) \] is moved to \( X(i,:) \) for \( i = 1, 2, \ldots, m \).

If \( \text{forwrd} \) is false, backward permutation:

\[ X(i,:) \] is moved to \( X(k[i-1,:]) \) for \( i = 1, 2, \ldots, m \).

Input Parameters

- **forwrd**
  - If \( \text{forwrd} \) is true, forward permutation.
  - If \( \text{forwrd} \) is false, backward permutation.

- **m**
  - The number of rows of the matrix \( X \). \( m \geq 0 \).

- **n**
  - The number of columns of the matrix \( X \). \( n \geq 0 \).

- **x**
  - Array, size at least \( \max(1, \ c x + n) \) for column major and \( \max(1, \ l d x + m) \) for row major layout. On entry, the \( m \)-by-\( n \) matrix \( X \).

- **ldx**
  - The leading dimension of the array \( X \). \( l d x \geq \max(1, m) \) for column major layout and \( l d x \geq \max(1, n) \) for row major layout.

- **k**
  - Array, size \( (m) \). On entry, \( k \) contains the permutation vector and is used as internal workspace.

Output Parameters

- **x**
  - On exit, \( x \) contains the permuted matrix \( X \).

- **k**
  - On exit, \( k \) is reset to its original value.

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.
If $info = -i$, the $i$-th parameter had an illegal value.
If $info = -1011$, memory allocation error occurred.

See Also

?lapmt
Performs a forward or backward permutation of the columns of a matrix.

Syntax

```c
lapack_int LAPACKE_slapmt (int matrix_layout, lapack_logical forwrd, lapack_int m, 
lapack_int n, float * x, lapack_int ldx, lapack_int * k);
lapack_int LAPACKE_dlapmt (int matrix_layout, lapack_logical forwrd, lapack_int m, 
lapack_int n, double * x, lapack_int ldx, lapack_int * k);
lapack_int LAPACKE_clapmt (int matrix_layout, lapack_logical forwrd, lapack_int m, 
lapack_int n, lapack_complex_float * x, lapack_int ldx, lapack_int * k);
lapack_int LAPACKE_zlapmt (int matrix_layout, lapack_logical forwrd, lapack_int m, 
lapack_int n, lapack_complex_double * x, lapack_int ldx, lapack_int * k);
```

Include Files

- mkl.h

Description

The routine ?lapmt rearranges the columns of the $m$-by-$n$ matrix $X$ as specified by the permutation $k[i - 1]$ for $i = 1, \ldots, n$.

If $forwrd \neq 0$, forward permutation:

$X(*,k(j))$ is moved to $X(*,j)$ for $j=1,2,\ldots,n$.

If $forwrd = 0$, backward permutation:

$X(*,j)$ is moved to $X(*,k(j))$ for $j=1,2,\ldots,n$.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **forwrd**: If $forwrd \neq 0$, forward permutation
  If $forwrd = 0$, backward permutation
- **m**: The number of rows of the matrix $X$. $m \geq 0$.
- **n**: The number of columns of the matrix $X$. $n \geq 0$.
- **x**: Array, size $ldx*n$. On entry, the $m$-by-$n$ matrix $X$.
- **ldx**: The leading dimension of the array $x$. $ldx \geq \max(1,m)$.
- **k**: Array, size ($n$). On entry, $k$ contains the permutation vector and is used as internal workspace.
**Output Parameters**

x  
On exit, x contains the permuted matrix X.

k  
On exit, k is reset to its original value.

**See Also**

?lapmr

?lapy2

*Returns* $\sqrt{x^2+y^2}$.

**Syntax**

float LAPACKE_slapy2 (float x, float y);
double LAPACKE_dlapy2 (double x, double y);

**Include Files**

- mkl.h

**Description**

The function ?lapy2 returns $\sqrt{x^2+y^2}$, avoiding unnecessary overflow or harmful underflow.

**Input Parameters**

x, y  
Specify the input values x and y.

**Return Values**

The function returns a value *val*.

If *val*=-1D0, the first argument was NaN.
If *val*=-2D0, the second argument was NaN.

?lapy3

*Returns* $\sqrt{x^2+y^2+z^2}$.

**Syntax**

float LAPACKE_slapy3 (float x, float y, float z);
double LAPACKE_dlapy3 (double x, double y, double z);

**Include Files**

- mkl.h

**Description**

The function ?lapy3 returns $\sqrt{x^2+y^2+z^2}$, avoiding unnecessary overflow or harmful underflow.

**Input Parameters**
Specify the input values x, y and z.

Return Values
This function returns a value val.
If val = -1D0, the first argument was NaN.
If val = -2D0, the second argument was NaN.
If val = -3D0, the third argument was NaN.

?laran
Returns a random real number from a uniform distribution.

Syntax
float slaran (lapack_int *iseed);
double dlaran (lapack_int *iseed);

Description
The ?laran routine returns a random real number from a uniform (0,1) distribution. This routine uses a multiplicative congruential method with modulus $2^{48}$ and multiplier 33952834046453. 48-bit integers are stored in four integer array elements with 12 bits per element. Hence the routine is portable across machines with integers of 32 bits or more.

Input Parameters
iseed
Array, size 4. On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and iseed[3] must be odd.

Output Parameters
iseed
On exit, the seed is updated.

Return Values
The function returns a random number.

?larfb
Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

```c
lapack_int LAPACKE_slarfb (int matrix_layout , char side , char trans , char direct ,
char storev , lapack_int m , lapack_int n , lapack_int k , const float * v , lapack_int ldv ,
const float * t , lapack_int ldt , float * c , lapack_int ldc );
```

```c
lapack_int LAPACKE_dlarfb (int matrix_layout , char side , char trans , char direct ,
char storev , lapack_int m , lapack_int n , lapack_int k , const double * v ,
lapack_int ldv , const double * t , lapack_int ldt , double * c , lapack_int ldc );
```

```c
lapack_int LAPACKE_clarfb (int matrix_layout , char side , char trans , char
```
direct, char storev, lapack_int m, lapack_int n, lapack_int k, const
lapack_complex_float * v, lapack_int ldv, const lapack_complex_float * t, lapack_int
ldt, lapack_complex_float * c, lapack_int ldc);
lapack_int LAPACKE_zlarfb (int matrix_layout, char side, char trans, char direct,
char storev, lapack_int m, lapack_int n, lapack_int k, const lapack_complex_double
* v, lapack_int ldv, const lapack_complex_double * t, lapack_int ldt,
lapack_complex_double * c, lapack_int ldc);

Include Files
- mkl.h

Description
The real flavors of the routine ?larfb apply a real block reflector \( H \) or its transpose \( H^T \) to a real \( m \)-by-\( n \)
matrix \( C \) from either left or right.
The complex flavors of the routine ?larfb apply a complex block reflector \( H \) or its conjugate transpose \( H^H \) to
a complex \( m \)-by-\( n \) matrix \( C \) from either left or right.

Input Parameters

\( \text{side} \)
If \( \text{side} = 'L' \): apply \( H \) or \( H^T \) for real flavors and \( H \) or \( H^H \) for complex
flavors from the left.
If \( \text{side} = 'R' \): apply \( H \) or \( H^T \) for real flavors and \( H \) or \( H^H \) for complex
flavors from the right.

\( \text{trans} \)
If \( \text{trans} = 'N' \): apply \( H \) (No transpose).
If \( \text{trans} = 'C' \): apply \( H^H \) (Conjugate transpose).
If \( \text{trans} = 'T' \): apply \( H^T \) (Transpose).

\( \text{direct} \)
Indicates how \( H \) is formed from a product of elementary reflectors
If \( \text{direct} = 'F' \): \( H = H(1)*H(2)*...*H(k) \) (forward)
If \( \text{direct} = 'B' \): \( H = H(k)*...*H(2)*H(1) \) (backward)

\( \text{storev} \)
Indicates how the vectors which define the elementary reflectors are
stored:
If \( \text{storev} = 'C' \): Column-wise
If \( \text{storev} = 'R' \): Row-wise

\( m \)
The number of rows of the matrix \( C \).

\( n \)
The number of columns of the matrix \( C \).

\( k \)
The order of the matrix \( T \) (equal to the number of elementary reflectors
whose product defines the block reflector).

\( v \)
The size limitations depend on values of parameters \( \text{storev} \) and \( \text{side} \) as
described in the following table:

| \( \text{storev} = C \) | \( \text{storev} = R \) |
The matrix \( v \). See Application Notes below.

### \( ldv \)

The leading dimension of the array \( v \). It should satisfy the following conditions:

<table>
<thead>
<tr>
<th>side</th>
<th>Column major</th>
<th>Row major</th>
</tr>
</thead>
<tbody>
<tr>
<td>( LD)</td>
<td>( \max(1, ldv \times k) )</td>
<td>( \max(1, ldv \times m) )</td>
</tr>
<tr>
<td>( RD)</td>
<td>( \max(1, ldv \times m) )</td>
<td>( \max(1, ldv \times n) )</td>
</tr>
</tbody>
</table>

### \( t \)

Array, size at least \( \max(1, ldt \times k) \).

Contains the triangular \( k \)-by-\( k \) matrix \( T \) in the representation of the block reflector.

### \( ldt \)

The leading dimension of the array \( t \).

\( ldt \geq k \).

### \( c \)

Array, size at least \( \max(1, ldc \times n) \) for column major layout and \( \max(1, ldc \times m) \) for row major layout.

On entry, the \( m \)-by-\( n \) matrix \( C \).

### \( ldc \)

The leading dimension of the array \( c \).

\( ldc \geq \max(1, m) \) for column major layout and \( ldc \geq \max(1, n) \) for row major layout.

#### Output Parameters

### \( c \)

On exit, \( c \) is overwritten by the product of the following:

- \( H^*C \) or \( H^*C \) or \( C^*H \) or \( C^*H^T \) for real flavors
- \( H^*C \) or \( H^*C \) or \( C^*H \) or \( C^*H^H \) for complex flavors

#### Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = -1011 \), memory allocation error occurred.

---

*Intel Math Kernel Library Developer Reference*
Application Notes

The shape of the matrix $V$ and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

$$
\begin{bmatrix}
1 & 1 \\
v_1 & v_2 & 1 \\
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3 \\
\end{bmatrix}
$$

$direct = 'F' and storev = 'C'$:  \quad $direct = 'F' and storev = 'R'$:

$$
\begin{bmatrix}
1 & v_1 & v_1 & v_1 \\
1 & v_2 & v_2 & v_2 \\
1 & v_3 & v_3 \\
\end{bmatrix}
$$

$direct = 'B' and storev = 'C'$:  \quad $direct = 'B' and storev = 'R'$:

$$
\begin{bmatrix}
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3 \\
1 & v_2 & v_3 \\
1 & v_3 \\
\end{bmatrix}
$$

?larfg

Generates an elementary reflector (Householder matrix).

**Syntax**

```c
lapack_int LAPACKE_slarfg (lapack_int n, float * alpha, float * x, lapack_int incx, float * tau);

lapack_int LAPACKE_dlarfg (lapack_int n, double * alpha, double * x, lapack_int incx, double * tau);

lapack_int LAPACKE_clarfg (lapack_int n, lapack_complex_float * alpha, lapack_complex_float * x, lapack_int incx, lapack_complex_float * tau);

lapack_int LAPACKE_zlarfg (lapack_int n, lapack_complex_double * alpha, lapack_complex_double * x, lapack_int incx, lapack_complex_double * tau);
```

**Include Files**

- `mkl.h`

**Description**

The routine ?larfg generates a real/complex elementary reflector $H$ of order $n$, such that
for real flavors and

\[
H^* \begin{bmatrix} \alpha \\ x \end{bmatrix} = \begin{bmatrix} \beta \\ 0 \end{bmatrix}, \quad H^*H = I,
\]
for complex flavors,

where \( \alpha \) and \( \beta \) are scalars (with \( \beta \) real for all flavors), and \( x \) is an \((n-1)\)-element real/complex vector. \( H \) is represented in the form

\[
H = I - \tau \begin{bmatrix} 1 \\ \nu^T \end{bmatrix} \quad \text{for real flavors and}
\]

\[
H = I - \tau \begin{bmatrix} 1 \\ \nu^H \end{bmatrix} \quad \text{for complex flavors},
\]

where \( \tau \) is a real/complex scalar and \( \nu \) is a real/complex \((n-1)\)-element vector, respectively. Note that for \text{clarfg/zlarfg}, \( H \) is not Hermitian.

If the elements of \( x \) are all zero (and, for complex flavors, \( \alpha \) is real), then \( \tau = 0 \) and \( H \) is taken to be the unit matrix.

Otherwise, \( 1 \leq \tau \leq 2 \) (for real flavors), or \( 1 \leq \text{Re}(\tau) \leq 2 \) and \( \text{abs}(\tau-1) \leq 1 \) (for complex flavors).

**Input Parameters**

- \( n \)  
  The order of the elementary reflector.

- \( \alpha \)  
  Array, size \((1+(n-2)\times\text{abs}(\text{incx}))\).
  On entry, the vector \( x \).

- \( \text{incx} \)  
  The increment between elements of \( x \). \( \text{incx} > 0 \).

**Output Parameters**

- \( \alpha \)  
  On exit, it is overwritten with the value \( \beta \).

- \( x \)  
  On exit, it is overwritten with the vector \( \nu \).

- \( \tau \)

**Return Values**

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} = -2 \), \( \alpha \) is NaN

If \( \text{info} = -3 \), array \( x \) contains NaN components.
?larft
Forms the triangular factor T of a block reflector H = I - V*T*V**H.

Syntax

```c
lapack_int LAPACKE_slarft (int matrix_layout, char direct, char storev, lapack_int n, lapack_int k, const float * v, lapack_int ldv, const float * tau, float * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_dlarft (int matrix_layout, char direct, char storev, lapack_int n, lapack_int k, const double * v, lapack_int ldv, const double * tau, double * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_clarft (int matrix_layout, char direct, char storev, lapack_int n, lapack_int k, const lapack_complex_float * v, lapack_int ldv, const lapack_complex_float * tau, lapack_complex_float * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_zlarft (int matrix_layout, char direct, char storev, lapack_int n, lapack_int k, const lapack_complex_double * v, lapack_int ldv, const lapack_complex_double * tau, lapack_complex_double * t, lapack_int ldt);
```

Include Files

- mkl.h

Description

The routine ?larft forms the triangular factor T of a real/complex block reflector H of order n, which is defined as a product of k elementary reflectors.

If `direct` = 'F', H = H(1)*H(2)*...*H(k) and T is upper triangular;

If `direct` = 'B', H = H(k)*...*H(2)*H(1) and T is lower triangular.

If `storev` = 'C', the vector which defines the elementary reflector H(i) is stored in the i-th column of the array v, and H = I - V*T*V (for real flavors) or H = I - V*H*V (for complex flavors).

If `storev` = 'R', the vector which defines the elementary reflector H(i) is stored in the i-th row of the array v, and H = I - V**T*T*V (for real flavors) or H = I - V**H*T*V (for complex flavors).

Input Parameters

- `direct` Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
  - '='F': H = H(1)*H(2)*...*H(k) (forward)
  - '='B': H = H(k)*...*H(2)*H(1) (backward)

- `storev` Specifies how the vectors which define the elementary reflectors are stored (see also Application Notes below):
  - '='C': column-wise
  - '='R': row-wise.

- `n` The order of the block reflector H. n ≥ 0.
The order of the triangular factor $T$ (equal to the number of elementary reflectors). $k \geq 1$.

The size limitations depend on values of parameters $\text{storev}$ and $\text{side}$ as described in the following table:

<table>
<thead>
<tr>
<th></th>
<th>$\text{storev} = C$</th>
<th>$\text{storev} = R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column major</td>
<td>$\max(1,ldv*k)$</td>
<td>$\max(1,ldv*n)$</td>
</tr>
<tr>
<td>Row major</td>
<td>$\max(1,ldv*n)$</td>
<td>$\max(1,ldv*k)$</td>
</tr>
</tbody>
</table>

The matrix $v$. See Application Notes below.

The leading dimension of the array $v$.

If $\text{storev} = 'C'$, $ldv \geq \max(1,n)$ for column major and $ldv \geq \max(1,k)$ for row major;

if $\text{storev} = 'R'$, $ldv \geq k$ for column major and $ldv \geq \max(1,n)$ for row major.

Array, size $(k)$. $\tau[i-1]$ must contain the scalar factor of the elementary reflector $H(i)$.

The leading dimension of the output array $t$. $ldt \geq k$.

**Output Parameters**

Array, size $ldt \times k$. The $k$-by-$k$ triangular factor $T$ of the block reflector. If $\text{direct} = 'F'$, $T$ is upper triangular; if $\text{direct} = 'B'$, $T$ is lower triangular. The rest of the array is not used.

The matrix $V$.

**Application Notes**

The shape of the matrix $V$ and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.

**direct = 'F' and storev = 'C':**

$$
\begin{bmatrix}
1 & 1 \\
v_1 & 1 \\
v_1 & v_2 & 1 \\
v_1 & v_2 & v_3 \\
v_1 & v_2 & v_3
\end{bmatrix}
$$

**direct = 'F' and storev = 'R':**

$$
\begin{bmatrix}
1 & v_1 & v_1 & v_1 \\
v_1 & v_2 & v_1 & v_1 \\
v_1 & v_2 & v_2 & v_1 \\
v_1 & v_2 & v_3 & v_1 \\
v_1 & v_2 & v_3 & v_1 \\
v_1 & v_2 & v_3 & v_2 \\
v_1 & v_2 & v_3 & v_3
\end{bmatrix}
$$
Applies an elementary reflector to a general rectangular matrix, with loop unrolling when the reflector has order less than or equal to 10.

**Syntax**

```c
lapack_int LAPACKE_slarfx (int matrix_layout, char side, lapack_int m, lapack_int n, const float * v, float tau, float * c, lapack_int ldc, float * work);

lapack_int LAPACKE_dlarfx (int matrix_layout, char side, lapack_int m, lapack_int n, const double * v, double tau, double * c, lapack_int ldc, double * work);

lapack_int LAPACKE_clarfx (int matrix_layout, char side, lapack_int m, lapack_int n, const lapack_complex_float * v, lapack_complex_float tau, lapack_complex_float * c, lapack_int ldc, lapack_complex_float * work);

lapack_int LAPACKE_zlarfx (int matrix_layout, char side, lapack_int m, lapack_int n, const lapack_complex_double * v, lapack_complex_double tau, lapack_complex_double * c, lapack_int ldc, lapack_complex_double * work);
```

**Include Files**

- mkl.h

**Description**

The routine ?larfx applies a real/complex elementary reflector $H$ to a real/complex $m$-by-$n$ matrix $C$, from either the left or the right.

$H$ is represented in the following forms:

- $H = I - \tau v^* v$, where $\tau$ is a real scalar and $v$ is a real vector.
- $H = I - \tau v^* v^T$, where $\tau$ is a complex scalar and $v$ is a complex vector.

If $\tau = 0$, then $H$ is taken to be the unit matrix.

**Input Parameters**

- **side**
  - If $\text{side} = 'L'$: form $H^* C$
  - If $\text{side} = 'R'$: form $C^* H$.

- **m**
  - The number of rows of the matrix $C$. 

\[ \begin{bmatrix} v_1 & v_2 & v_3 \\ v_1 & v_2 & v_3 \\ 1 & v_2 & v_3 \\ 1 & v_3 \\ 1 \end{bmatrix} \]
The number of columns of the matrix $C$.

$\nu$

Array, size

$(m)$ if $side = 'L'$ or

$(n)$ if $side = 'R'$.

The vector $v$ in the representation of $H$.

$\tau$

The value $\tau$ in the representation of $H$.

$c$

Array, size at least max(1, $ldc*n$) for column major layout and max (1, $ldc*n$) for row major layout. On entry, the $m$-by-$n$ matrix $C$.

$\text{ldc}$

The leading dimension of the array $c$. $\text{ldc} \geq (1,m)$.

$\text{work}$

Workspace array, size

$(n)$ if $side = 'L'$ or

$(m)$ if $side = 'R'$.

$\text{work}$ is not referenced if $H$ has order < 11.

**Output Parameters**

$c$

On exit, $C$ is overwritten by the matrix $H*C$ if $side = 'L'$, or $C*H$ if $side = 'R'$.

**?large**

*Pre- and post-multiplies a real general matrix with a random orthogonal matrix.*

**Syntax**

```c
void slarge (lapack_int *n, float *a, lapack_int *lda, lapack_int *iseed, float * work, lapack_int *info);
void dlarge (lapack_int *n, double *a, lapack_int *lda, lapack_int *iseed, double * work, lapack_int *info);
void clarge (lapack_int *n, lapack_complex *a, lapack_int *lda, lapack_int *iseed, lapack_complex * work, lapack_int *info);
void zlarge (lapack_int *n, lapack_complex_double *a, lapack_int *lda, lapack_int *iseed, lapack_complex_double * work, lapack_int *info);
```

**Include Files**

- mkl.h

**Description**

The routine ?large pre- and post-multiplies a general $n$-by-$n$ matrix $A$ with a random orthogonal or unitary matrix: $A = U*D*U^\top$.

**Input Parameters**

$n$

The order of the matrix $A$. $n \geq 0$
Array, size \( ld a \) by \( n \).
On entry, the original \( n \)-by-\( n \) matrix \( A \).

\( lda \)
The leading dimension of the array \( a \). \( lda \geq n \).

\( iseed \)
Array, size 4.
On entry, the seed of the random number generator. The array elements must be between 0 and 4095, and \( iseed[3] \) must be odd.

Workspace array, size \( 2n \).

\section*{Output Parameters}

On exit, \( A \) is overwritten by \( U^*A^*U' \) for some random orthogonal matrix \( U \).

On exit, the seed is updated.

If \( info = 0 \), the execution is successful.
If \( info < 0 \), the \( i \)-th parameter had an illegal value.

\texttt{?larnd}

\textit{Returns a random real number from a uniform or normal distribution.}

\textbf{Syntax}

\begin{verbatim}
float slarnd (lapack_int *idist, lapack_int *iseed);
double dlarnd (lapack_int *idist, lapack_int *iseed);
\end{verbatim}

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

For non-gfortran (libmkl_intel_*) interface libraries:

\begin{verbatim}
void clarnd (lapack_complex_float *res, lapack_int *idist, lapack_int *iseed);
void zlarnd (lapack_complex_double *res, lapack_int *idist, lapack_int *iseed);
\end{verbatim}

For gfortran (libmkl_gf_*) interface libraries:

\begin{verbatim}
lapack_complex_float clarnd (lapack_int *idist, lapack_int *iseed);
lapack_complex_double zlarnd (lapack_int *idist, lapack_int *iseed);
\end{verbatim}

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the \textit{Intel® MKL Developer Guide}.

\textbf{Include Files}

\begin{itemize}
  \item mkl.h
\end{itemize}

\textbf{Description}

The routine \texttt{?larnd} returns a random number from a uniform or normal distribution.

\textbf{Input Parameters}

\texttt{idist}

\textit{Specifies the distribution of the random numbers. For \texttt{slarnd} and \texttt{dlarnd}:}
= 1: uniform (0,1)
= 2: uniform (-1,1)
= 3: normal (0,1).

For clarnd and zlanrd:
= 1: real and imaginary parts each uniform (0,1)
= 2: real and imaginary parts each uniform (-1,1)
= 3: real and imaginary parts each normal (0,1)
= 4: uniformly distributed on the disc abs(z) ≤ 1
= 5: uniformly distributed on the circle abs(z) = 1

iseed
Array, size 4.

On entry, the seed of the random number generator. The array elements
must be between 0 and 4095, and iseed[3] must be odd.

Output Parameters

iseed
On exit, the seed is updated.

Return Values

The function returns a random number (for complex variations libmkl_gf_*
interface layer/libraries return
the result as the parameter res).

?larnv
Returns a vector of random numbers from a uniform
or normal distribution.

Syntax

lapack_int LAPACKE_slarnv (lapack_int idist , lapack_int * iseed , lapack_int n , float
* x );
lapack_int LAPACKE_dlarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
double * x );
lapack_int LAPACKE_clarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
lapack_complex_float * x );
lapack_int LAPACKE_zlarnv (lapack_int idist , lapack_int * iseed , lapack_int n ,
lapack_complex_double * x );

Include Files

• mkl.h

Description

The routine ?larnv returns a vector of n random real/complex numbers from a uniform or normal
distribution.

This routine calls the auxiliary routine ?laruv to generate random real numbers from a uniform (0,1)
distribution, in batches of up to 128 using vectorisable code. The Box-Muller method is used to transform
numbers from a uniform to a normal distribution.
Input Parameters

\textbf{idist}

Specifies the distribution of the random numbers: for \texttt{slarnv} and \texttt{dlarnv}:

\begin{itemize}
  \item $= 1$: uniform (0,1)
  \item $= 2$: uniform (-1,1)
  \item $= 3$: normal (0,1).
\end{itemize}

for \texttt{clarnv} and \texttt{zlarnv}:

\begin{itemize}
  \item $= 1$: real and imaginary parts each uniform (0,1)
  \item $= 2$: real and imaginary parts each uniform (-1,1)
  \item $= 3$: real and imaginary parts each normal (0,1)
  \item $= 4$: uniformly distributed on the disc $\text{abs}(z) < 1$
  \item $= 5$: uniformly distributed on the circle $\text{abs}(z) = 1$
\end{itemize}

\textbf{iseed}

Array, size (4).

On entry, the seed of the random number generator; the array elements must be between 0 and 4095, and \texttt{iseed(4)} must be odd.

\textbf{n}

The number of random numbers to be generated.

Output Parameters

\textbf{x}

Array, size (n). The generated random numbers.

\textbf{iseed}

On exit, the seed is updated.

Return Values

This function returns a value \texttt{info}.

If $\texttt{info} = 0$, the execution is successful.

\textbf{?laror}

Pre- or post-multiplies an m-by-n matrix by a random orthogonal/unitary matrix.

\textbf{Syntax}

\begin{verbatim}
void slaror (char *side, char *init, lapack_int *m, lapack_int *n, float *a, lapack_int *ida, lapack_int *iseed, float *x, lapack_int *info);
void dlaror (char *side, char *init, lapack_int *m, lapack_int *n, double *a, lapack_int *ida, lapack_int *iseed, double *x, lapack_int *info);
void claror (char *side, char *init, lapack_int *m, lapack_int *n, lapack_complex *a, lapack_int *ida, lapack_int *iseed, lapack_complex *x, lapack_int *info);
void zlaror (char *side, char *init, lapack_int *m, lapack_int *n, lapack_complex_double *a, lapack_int *ida, lapack_int *iseed, lapack_complex_double *x, lapack_int *info);
\end{verbatim}

\textbf{Include Files}

- mkl.h
Description

The routine ?laror pre- or post-multiplies an \( m \)-by-\( n \) matrix \( A \) by a random orthogonal or unitary matrix \( U \), overwriting \( A \). \( A \) may optionally be initialized to the identity matrix before multiplying by \( U \). \( U \) is generated using the method of G.W. Stewart (SIAM J. Numer. Anal. 17, 1980, 403-409).

Input Parameters

\textit{side}

Specifies whether \( A \) is multiplied by \( U \) on the left or right.

for \texttt{slaror} and \texttt{dlaror}:
- If \texttt{side} = 'L', multiply \( A \) on the left (premultiply) by \( U \).
- If \texttt{side} = 'R', multiply \( A \) on the right (postmultiply) by \( U^T \).
- If \texttt{side} = 'C' or 'T', multiply \( A \) on the left by \( U \) and the right by \( U^T \).

for \texttt{claror} and \texttt{zlaror}:
- If \texttt{side} = 'L', multiply \( A \) on the left (premultiply) by \( U \).
- If \texttt{side} = 'R', multiply \( A \) on the right (postmultiply) by \( U^C \).
- If \texttt{side} = 'C', multiply \( A \) on the left by \( U \) and the right by \( U^C \).
- If \texttt{side} = 'T', multiply \( A \) on the left by \( U \) and the right by \( U^T \).

\textit{init}

Specifies whether or not \( a \) should be initialized to the identity matrix.

- If \texttt{init} = 'I', initialize \( a \) to (a section of) the identity matrix before applying \( U \).
- If \texttt{init} = 'N', no initialization. Apply \( U \) to the input matrix \( A \).
- \texttt{init} = 'I' generates square or rectangular orthogonal matrices:
  - For \( m = n \) and \texttt{side} = 'L' or 'R', the rows and the columns are orthogonal to each other.
  - For rectangular matrices where \( m < n \):
    - If \texttt{side} = 'R', \texttt{slaror} produces a dense matrix in which rows are orthogonal and columns are not.
    - If \texttt{side} = 'L', \texttt{slaror} produces a matrix in which rows are orthogonal, first \( m \) columns are orthogonal, and remaining columns are zero.
  - For rectangular matrices where \( m > n \):
    - If \texttt{side} = 'L', \texttt{slaror} produces a dense matrix in which columns are orthogonal and rows are not.
    - If \texttt{side} = 'R', \texttt{slaror} produces a matrix in which columns are orthogonal, first \( m \) rows are orthogonal, and remaining rows are zero.

\textit{m}

The number of rows of \( A \).

\textit{n}

The number of columns of \( A \).

\textit{a}

Array, size \( \text{lda} \) by \( n \).

\textit{lda}

The leading dimension of the array \( a \).

\[ \text{lda} \geq \max(1, m) \]

\textit{iseed}

Array, size (4).
On entry, specifies the seed of the random number generator. The array elements must be between 0 and 4095; if not they are reduced mod 4096. Also, iseed[3] must be odd.

Workspace array, size \((3 \max(m, n))\).

<table>
<thead>
<tr>
<th>Value of side</th>
<th>Length of workspace</th>
</tr>
</thead>
<tbody>
<tr>
<td>'L'</td>
<td>(2m + n)</td>
</tr>
<tr>
<td>'R'</td>
<td>(2n + m)</td>
</tr>
<tr>
<td>'C' or 'T'</td>
<td>(3n)</td>
</tr>
</tbody>
</table>

Output Parameters

**a**

On exit, overwritten

by \(UA\) (if side = 'L'),

by \(AU\) (if side = 'R'),

by \(UAU^T\) (if side = 'C' or 'T').

**iseed**

The values of iseed are changed on exit, and can be used in the next call to continue the same random number sequence.

**info**

Array, size (4).

For slarot and dlaror:

If info = 0, the execution is successful.

If info < 0, the \(i\)-th parameter had an illegal value.

If info = 1, the random numbers generated by ?larot are bad.

For clarot and zlarot:

If info = 0, the execution is successful.

If info = -1, side is not 'L', 'R', 'C', or 'T'.

If info = -3, if \(m\) is negative.

If info = -4, if \(m\) is negative or if side is 'C' or 'T' and \(n\) is not equal to \(m\).

If info = -6, if lda is less than \(m\).

?larot

Applies a Givens rotation to two adjacent rows or columns.

**Syntax**

```c
void slarot (lapack_logical *lrows, lapack_logical * ileft, lapack_logical * iright, lapack_int * nl, float * c, float * s, float * a, lapack_int * lda, float * xleft, float * xright);
void dlarot (lapack_logical * lrows, lapack_logical * ileft, lapack_logical * iright, lapack_int * nl, double * c, double * s, double * a, lapack_int * lda, double * xleft, double * xright);
```
void clarot (lapack_logical *lrows, lapack_logical *illeft, lapack_logical *iright, lapack_int *nl, lapack_complex *c, lapack_complex *s, lapack_complex *a, lapack_int *lda, lapack_complex *xleft, lapack_complex *xright);

void zlarot (lapack_logical *lrows, lapack_logical *illeft, lapack_logical *iright, lapack_int *nl, lapack_complex_double *c, lapack_complex_double *s, lapack_complex_double *a, lapack_int *lda, lapack_complex_double *xleft, lapack_complex_double *xright);

Include Files
• mkl.h

Description
The routine ?larot applies a Givens rotation to two adjacent rows or columns, where one element of the first or last column or row is stored in some format other than GE so that elements of the matrix may be used or modified for which no array element is provided.

One example is a symmetric matrix in SB format (bandwidth = 4), for which uplo = 'L'. Two adjacent rows will have the format:

row j: C > C > C > C > . . . .
row j + 1: C > C > C > C > . . . .

'*' indicates elements for which storage is provided.
'. ' indicates elements for which no storage is provided, but are not necessarily zero; their values are determined by symmetry.
' ' indicates elements which are required to be zero, and have no storage provided.

Those columns which have two '***' entries can be handled by srot (for slarot and clarot), or by drot (for dlarot and zlarot).

Those columns which have no '***' entries can be ignored, since as long as the Givens rotations are carefully applied to preserve symmetry, their values are determined.

Those columns which have one '***' have to be handled separately, by using separate variables p and q:

row j: C > C > C > C > C > p . . . .
row j + 1: q C > C > C > C > . . . .

If element p is set correctly, ?larot rotates the column and sets p to its new value. The next call to ?larot rotates columns j and j +1, and restore symmetry. The element q is zero at the beginning, and non-zero after the rotation. Later, rotations would presumably be chosen to zero q out.

Typical Calling Sequences: rotating the i-th and (i +1)-st rows.

Input Parameters
lrows
If lrows = 1, ?larot rotates two rows.
If lrows = 0, ?larot rotates two columns.

illeft
If illeft = 1, xleft is used instead of the corresponding element of a for the first element in the second row (if lrows = 0) or column (if lrows=1).
If illeft = 0, the corresponding element of a is used.

iright
If \( lleft = 1 \), \( xright \) is used instead of the corresponding element of \( a \) for the first element in the second row (if \( lrows = 0 \)) or column (if \( lrows = 1 \)).

If \( lright = 0 \), the corresponding element of \( a \) is used.

\( nl \)

The length of the rows (if \( lrows = 1 \)) or columns (if \( lrows = 1 \)) to be rotated.

If \( xleft \) or \( xright \) are used, the columns or rows they are in should be included in \( nl \), e.g., if \( lleft = lright = 1 \), then \( nl \) must be at least 2.

The number of rows or columns to be rotated exclusive of those involving \( xleft \) and/or \( xright \) may not be negative, i.e., \( nl \) minus how many of \( lleft \) and \( lright \) are 1 must be at least zero; if not, xerbla is called.

\( c, s \)

Specify the Givens rotation to be applied.

If \( lrows = 1 \), then the matrix

\[
\begin{bmatrix}
C & S \\
-S & C
\end{bmatrix}
\]

is applied from the left.

If \( lrows = 0 \), then the transpose thereof is applied from the right.

\( a \)

The array containing the rows or columns to be rotated. The first element of \( a \) should be the upper left element to be rotated.

\( lda \)

The "effective" leading dimension of \( a \).

If \( a \) contains a matrix stored in GE or SY format, then this is just the leading dimension of \( A \).

If \( a \) contains a matrix stored in band (GB or SB) format, then this should be one less than the leading dimension used in the calling routine. Thus, if \( a \) in \( ?larot \) is of size \( lda \times n \), then \( a[(j - 1) \times lda] \) would be the \( j \)-th element in the first of the two rows to be rotated, and \( a[(j - 1) \times lda + 1] \) would be the \( j \)-th in the second, regardless of how the array may be stored in the calling routine. \( a \) cannot be dimensioned, because for band format the row number may exceed \( lda \), which is not legal FORTRAN.

If \( lrows = 1 \), then \( lda \) must be at least 1, otherwise it must be at least \( nl \) minus the number of 1 values in \( xleft \) and \( xright \).

\( xleft \)

If \( lrows = 1 \), \( xleft \) is used and modified instead of \( a[1] \) (if \( lrows = 1 \)) or \( a[lda + 1] \) (if \( lrows = 0 \)).

\( xright \)

If \( lright = 1 \), \( xright \) is used and modified instead of \( a[(nl - 1) \times lda] \) (if \( lrows = 1 \)) or \( a[nl - 1] \) (if \( lrows = 0 \)).

**Output Parameters**

\( a \)

On exit, modified array \( A \).
?lartgp
Generates a plane rotation.

Syntax
lapack_int LAPACKE_slartgp (float f, float g, float* cs, float* sn, float* r);
lapack_int LAPACKE_dlartgp (double f, double g, double* cs, double* sn, double* r);

Include Files
• mkl.h

Description
The routine generates a plane rotation so that

\[ \begin{bmatrix} cs & sn \\ -sn & cs \end{bmatrix} \cdot \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \]

where \( cs^2 + sn^2 = 1 \)

This is a slower, more accurate version of the BLAS Level 1 routine ?rotg, except for the following differences:

• \( f \) and \( g \) are unchanged on return.
• If \( g = 0 \), then \( cs = (+/-)1 \) and \( sn = 0 \).
• If \( f = 0 \) and \( g \neq 0 \), then \( cs = 0 \) and \( sn = (+/-)1 \).

The sign is chosen so that \( r \geq 0 \).

Input Parameters
\( f, g \)  
The first and second component of the vector to be rotated.

Output Parameters
\( cs \)  
The cosine of the rotation.

\( sn \)  
The sine of the rotation.

\( r \)  
The nonzero component of the rotated vector.

Return Values
If \( info = 0 \), the execution is successful.
If \( info = -1 \), \( f \) is NaN.
If \( info = -2 \), \( g \) is NaN.

See Also
?rotg
?lartgs
?lartgs
Generates a plane rotation designed to introduce a bulge in implicit QR iteration for the bidiagonal SVD problem.

Syntax
lapack_int LAPACKE_slartgs (float x, float y, float sigma, float* cs, float* sn);
lapack_int LAPACKE_dlartgs (double x, double y, double sigma, double* cs, double* sn);

Include Files
• mkl.h

Description
The routine generates a plane rotation designed to introduce a bulge in Golub-Reinsch-style implicit QR iteration for the bidiagonal SVD problem. \( x \) and \( y \) are the top-row entries, and \( \sigma \) is the shift. The computed \( cs \) and \( sn \) define a plane rotation that satisfies the following:

\[
\begin{bmatrix}
\cos \theta & \sin \theta \\
-\sin \theta & \cos \theta \\
\end{bmatrix}
\begin{bmatrix}
    x^2 - \sigma \\
    x \cdot y \\
\end{bmatrix}
= \begin{bmatrix}
    r \\
    0 \\
\end{bmatrix}
\]

with \( r \) nonnegative.
If \( x^2 - \sigma \) and \( x \cdot y \) are 0, the rotation is by \( \pi/2 \).

Input Parameters
\( x, y \) The (1,1) and (1,2) entries of an upper bidiagonal matrix, respectively.
\( \sigma \) Shift

Output Parameters
\( cs \) The cosine of the rotation.
\( sn \) The sine of the rotation.

Return Values
If \( info = 0 \), the execution is successful.
If \( info = -1 \), \( x \) is NaN.
If \( info = -2 \), \( y \) is NaN.
If \( info = -3 \), \( \sigma \) is NaN.

See Also
?lartgp

?lascl
Multiplies a general rectangular matrix by a real scalar defined as \( c_{to}/c_{from} \).
Syntax

```c
lapack_int LAPACKE_slascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
float cfrom, float cto, lapack_int m, lapack_int n, float * a, lapack_int lda);

lapack_int LAPACKE_dlascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
double cfrom, double cto, lapack_int m, lapack_int n, double * a, lapack_int lda);

lapack_int LAPACKE_clascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
float cfrom, float cto, lapack_int m, lapack_int n, lapack_complex_float * a,
lapack_int lda);

lapack_int LAPACKE_zlascl (int matrix_layout, char type, lapack_int kl, lapack_int ku,
double cfrom, double cto, lapack_int m, lapack_int n, lapack_complex_double * a,
lapack_int lda);
```

Include Files

- mkl.h

Description

The routine **?lascl** multiplies the \( m \)-by-\( n \) real/complex matrix \( A \) by the real scalar \( cto/cfrom \). The operation is performed without over/underflow as long as the final result \( cto \times A(i,j)/cfrom \) does not over/underflow. The **type** specifies that \( A \) may be full, upper triangular, lower triangular, upper Hessenberg, or banded.

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **type**
  This parameter specifies the storage type of the input matrix.

  - 'G': \( A \) is a full matrix.
  - 'L': \( A \) is a lower triangular matrix.
  - 'U': \( A \) is an upper triangular matrix.
  - 'H': \( A \) is an upper Hessenberg matrix.
  - 'B': \( A \) is a symmetric band matrix with lower bandwidth \( kl \) and upper bandwidth \( ku \) and with the only the lower half stored.
  - 'Q': \( A \) is a symmetric band matrix with lower bandwidth \( kl \) and upper bandwidth \( ku \) and with the only the upper half stored.
  - 'Z': \( A \) is a band matrix with lower bandwidth \( kl \) and upper bandwidth \( ku \). See description of the ?gbtrf function for storage details.

- **kl**
  The lower bandwidth of \( A \). Referenced only if **type** = 'B', 'Q' or 'Z'.

- **ku**
  The upper bandwidth of \( A \). Referenced only if **type** = 'B', 'Q' or 'Z'.

- **cfrom, cto**
  The matrix \( A \) is multiplied by \( cto/cfrom \). \( A(i,j) \) is computed without over/underflow if the final result \( cto \times A(i,j)/cfrom \) can be represented without over/underflow. \( cfrom \) must be nonzero.

- **m**
  The number of rows of the matrix \( A \). \( m \geq 0 \).

- **n**
  The number of columns of the matrix \( A \). \( n \geq 0 \).
Input Parameters

\( n \)

On entry, the row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array \( d \).

\( \text{sqre} \)

Specifies the column dimension of the bidiagonal matrix.

- If \( \text{sqre} = 0 \): the bidiagonal matrix has column dimension \( m = n \).
- If \( \text{sqre} = 1 \): the bidiagonal matrix has column dimension \( m = n+1 \).

Output Parameters

\( a \)

The multiplied matrix \( A \).

\( \text{info} \)

- If \( \text{info} = 0 \): successful exit
- If \( \text{info} = -i < 0 \), the \( i \)-th argument had an illegal value.

See Also

- \?gbtrf

\?lasd0

Computes the singular values of a real upper bidiagonal \( n \)-by-\( m \) matrix \( B \) with diagonal \( d \) and off-diagonal \( e \). Used by \?bdsdc.

Syntax

```c
void slasd0( lapack_int *n, lapack_int *sqre, float *d, float *e, float *u, lapack_int *ldu, float *vt, lapack_int *ldvt, lapack_int *smlsiz, lapack_int *iwork, float *work, lapack_int *info);
void dlasd0( lapack_int *n, lapack_int *sqre, double *d, double *e, double *u, lapack_int *ldu, double *vt, lapack_int *ldvt, lapack_int *smlsiz, lapack_int *iwork, double *work, lapack_int *info);
```

Include Files

- mkl.h

Description

Using a divide and conquer approach, the routine \?lasd0 computes the singular value decomposition (SVD) of a real upper bidiagonal \( n \)-by-\( m \) matrix \( B \) with diagonal \( d \) and offdiagonal \( e \), where \( m = n + \text{sqre} \).

The algorithm computes orthogonal matrices \( U \) and \( VT \) such that \( B = U^*S^*VT \). The singular values \( S \) are overwritten on \( d \).

The related subroutine \?lasda computes only the singular values, and optionally, the singular vectors in compact form.

Input Parameters

\( \text{lda} \)

The leading dimension of the array \( a \).

\( \text{lda} \geq \max(1, m) \).
Array, DIMENSION (n). On entry, d contains the main diagonal of the bidiagonal matrix.

e
Array, DIMENSION (m-1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, e is destroyed.

ldu
On entry, leading dimension of the output array u.

ldvt
On entry, leading dimension of the output array vt.

smlsiz
On entry, maximum size of the subproblems at the bottom of the computation tree.

iwork
Workspace array, dimension must be at least (8n).

work
Workspace array, dimension must be at least (3m^2+2m).

Output Parameters

d
On exit d, If info = 0, contains singular values of the bidiagonal matrix.

Array, DIMENSION at least (ldq, n). On exit, u contains the left singular vectors.

vt
Array, DIMENSION at least (ldvt, m). On exit, vt^T contains the right singular vectors.

info
If info = 0: successful exit.
If info = -i < 0, the i-th argument had an illegal value.
If info = 1, a singular value did not converge.

?lasd1
Computes the SVD of an upper bidiagonal matrix B of the specified size. Used by ?bdsdc.

Syntax

void slasd1( lapack_int *nl, lapack_int *nr, lapack_int *sqre, float *d, float *alpha, float *beta, float *u, lapack_int *ldu, float *vt, lapack_int *ldvt, lapack_int *idxq, lapack_int *iwork, float *work, lapack_int *info );

void dlasd1( lapack_int *nl, lapack_int *nr, lapack_int *sqre, double *d, double *alpha, double *beta, double *u, lapack_int *ldu, double *vt, lapack_int *ldvt, lapack_int *idxq, lapack_int *iwork, double *work, lapack_int *info );

Include Files

• mkl.h

Description

The routine computes the SVD of an upper bidiagonal n-by-m matrix B, where n = nl + nr + 1 and m = n + sqre.

The routine ?lasd1 is called from ?lasd0.
A related subroutine `?lasd7` handles the case in which the singular values (and the singular vectors in factored form) are desired.

`?lasdi` computes the SVD as follows:

\[
VT = U(i\!n)^* \begin{bmatrix}
D(i\!n) & 0 & 0 \\
0 & a & 0 \\
0 & 0 & D2(i\!n)
\end{bmatrix} VT(i\!n)
\]

\[
= U(out)^*(D(out) 0)*VT(out)
\]

where \(Z^T = (Z1^T a Z2^T b) = u^T VT^T\), and \(u\) is a vector of dimension \(m\) with \(alpha\) and \(beta\) in the \(n\ell+1\) and \(n\ell+2\)-th entries and zeros elsewhere; and the entry \(b\) is empty if \(sqre = 0\).

The left singular vectors of the original matrix are stored in \(u\), and the transpose of the right singular vectors are stored in \(vT\), and the singular values are in \(d\). The algorithm consists of three stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or when there are zeros in the \(Z\) vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine `?lasd2`.
2. The second stage consists of calculating the updated singular values. This is done by finding the square roots of the roots of the secular equation via the routine `?lasd4` (as called by `?lasd3`). This routine also calculates the singular vectors of the current problem.
3. The final stage consists of computing the updated singular vectors directly using the updated singular values. The singular vectors for the current problem are multiplied with the singular vectors from the overall problem.

**Input Parameters**

- \(nl\)
  - The row dimension of the upper block.
  - \(nl\geq 1\).

- \(nr\)
  - The row dimension of the lower block.
  - \(nr\geq 1\).

- \(sqre\)
  - If \(sqre = 0\): the lower block is an \(nr\)-by-\(nr\) square matrix.
  - If \(sqre = 1\): the lower block is an \(nr\)-by-\((nr+1)\) rectangular matrix. The bidiagonal matrix has row dimension \(n = nl + nr + 1\), and column dimension \(m = n + sqre\).

- \(d\)
  - Array, DIMENSION \((nl+nr+1)\). \(d(1:nl, 1:nl)\) contains the singular values of the upper block; and \(d(nl+2:n)\) contains the singular values of the lower block.

- \(alpha\)
  - Contains the diagonal element associated with the added row.

- \(beta\)
  - Contains the off-diagonal element associated with the added row.

- \(u\)
  - Array, DIMENSION \((ldu, n)\). \(u(1:nl, 1:nl)\) contains the left singular vectors of the upper block; \(u(nl+2:n, nl+2:n)\) contains the left singular vectors of the lower block.

- \(ldu\)
  - The leading dimension of the array \(U\).
  - \(ldu\geq \max(1, n)\).
vt

Array, DIMENSION (ldvt, m), where m = n + sqre.

On entry vt(1:nl+1, 1:nl+1)^T contains the right singular vectors of the upper block; vt(nl+2:m, nl+2:m)^T contains the right singular vectors of the lower block.

ldvt

The leading dimension of the array vt.

ldvt >= max(1, M).

iwork

Workspace array, DIMENSION (4n).

work

Workspace array, DIMENSION (3m^2 + 2m).

Output Parameters

d

On exit d(1:n) contains the singular values of the modified matrix.

alpha

On exit, the diagonal element associated with the added row deflated by max( abs( alpha ), abs( beta ), abs( D(I) ) ), I = 1,n.

beta

On exit, the off-diagonal element associated with the added row deflated by max( abs( alpha ), abs( beta ), abs( D(I) ) ), I = 1,n.

u

On exit u contains the left singular vectors of the bidiagonal matrix.

vt

On exit vt^T contains the right singular vectors of the bidiagonal matrix.

idxq

Array, DIMENSION (n). Contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, d(idxq( i = 1, n )) will be in ascending order.

info

If info = 0: successful exit.
If info = -i < 0, the i-th argument had an illegal value.
If info = 1, a singular value did not converge.

?lasd2

Merges the two sets of singular values together into a single sorted set. Used by ?bdsdc.

Syntax

void slasd2( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, float *d, float *z, float *alpha, float *beta, float *u, lapack_int *ldu, float *vt, lapack_int *ldvt, float *dsigma, float *u2, lapack_int *ldu2, float *vt2, lapack_int *ldvt2, lapack_int *idxp, lapack_int *idx, lapack_int *idxq, lapack_int *coltyp, lapack_int *info );

void dlasd2( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, double *d, double *z, double *alpha, double *beta, double *u, lapack_int *ldu, double *vt, lapack_int *ldvt, double *dsigma, double *u2, lapack_int *ldu2, double *vt2, lapack_int *ldvt2, lapack_int *idxp, lapack_int *idx, lapack_int *idxq, lapack_int *coltyp, lapack_int *info );
Include Files
- mkl.h

Description
The routine ?lasd2 merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one. The routine ?lasd2 is called from ?lasd1.

Input Parameters

- \( nl \) 
  The row dimension of the upper block. \( nl \geq 1 \).

- \( nr \) 
  The row dimension of the lower block. \( nr \geq 1 \).

- \( sqre \) 
  If \( sqre = 0 \): the lower block is an \( nr \)-by-\( nr \) square matrix.
  If \( sqre = 1 \): the lower block is an \( nr \)-by-(\( nr \)+1) rectangular matrix. The bidiagonal matrix has \( n = nl + nr + 1 \) rows and \( m = n + sqre \geq n \) columns.

- \( d \) 
  Array, DIMENSION (\( n \)). On entry \( d \) contains the singular values of the two submatrices to be combined.

- \( alpha \) 
  Contains the diagonal element associated with the added row.

- \( beta \) 
  Contains the off-diagonal element associated with the added row.

- \( u \) 
  Array, DIMENSION (\( ldu, n \)). On entry \( u \) contains the left singular vectors of two submatrices in the two square blocks with corners at \((1,1), (nl, nl)\), and \((nl+2, nl+2), (n,n)\).

- \( ldu \) 
  The leading dimension of the array \( u \). \( ldu \geq n \).

- \( ldu2 \) 
  The leading dimension of the output array \( u2 \). \( ldu2 \geq n \).

- \( vt \) 
  Array, DIMENSION (\( ldvt, m \)). On entry, \( vt^T \) contains the right singular vectors of two submatrices in the two square blocks with corners at \((1,1), (nl+1, nl+1)\), and \((nl+2, nl+2), (m,m)\).

- \( ldvt \) 
  The leading dimension of the array \( vt \). \( ldvt \geq m \).

- \( ldvt2 \) 
  The leading dimension of the output array \( vt2 \). \( ldvt2 \geq m \).

- \( idxp \) 
  Workspace array, DIMENSION (\( n \)). This will contain the permutation used to place deflated values of \( D \) at the end of the array. On output \( idxp(2:k) \) points to the nondeflated \( d \)-values and \( idxp(k+1:n) \) points to the deflated singular values.

- \( idx \) 
  Workspace array, DIMENSION (\( n \)). This will contain the permutation used to sort the contents of \( d \) into ascending order.
**Workspace array, DIMENSION (n).** As workspace, this array contains a label that indicates which of the following types a column in the u2 matrix or a row in the vt2 matrix is:

1 : non-zero in the upper half only
2 : non-zero in the lower half only
3 : dense
4 : deflated.

**idxq Array, DIMENSION (n).** This parameter contains the permutation that separately sorts the two sub-problems in D in the ascending order. Note that entries in the first half of this permutation must first be moved one position backwards and entries in the second half must have \(nl+1\) added to their values.

### Output Parameters

**k**
Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. \(1 \leq k \leq n\).

**d**
On exit D contains the trailing \((n-k)\) updated singular values (those which were deflated) sorted into increasing order.

**u**
On exit u contains the trailing \((n-k)\) updated left singular vectors (those which were deflated) in its last \(n-k\) columns.

**z**
Array, DIMENSION (n). On exit, z contains the updating row vector in the secular equation.

**dsigma**
Array, DIMENSION (ldu2, n). Contains a copy of the first \(k-1\) left singular values and one zero) in the secular equation.

**u2**
Array, DIMENSION (ldu2, n). Contains a copy of the first \(k-1\) left singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new left singular vectors. u2 is arranged into four blocks. The first block contains a column with 1 at \(nl+1\) and zero everywhere else; the second block contains non-zero entries only at and above \(nl\); the third contains non-zero entries only below \(nl+1\); and the fourth is dense.

**vt**
On exit, vt^T contains the trailing \((n-k)\) updated right singular vectors (those which were deflated) in its last \(n-k\) columns. In case \(sqre = 1\), the last row of vt spans the right null space.

**vt2**
Array, DIMENSION (ldvt2, n). vt2^T contains a copy of the first \(k\) right singular vectors which will be used by ?lasd3 in a matrix multiply (?gemm) to solve for the new right singular vectors. vt2 is arranged into three blocks. The first block contains a row that corresponds to the special 0 diagonal element in sigma; the second block contains non-zeros only at and before \(nl +1\); the third block contains non-zeros only at and after \(nl +2\).

**idxc**
Array, DIMENSION (n). This will contain the permutation used to arrange the columns of the deflated u matrix into three groups: the first group contains non-zero entries only at and above \(nl\), the second contains non-zero entries only below \(nl+2\), and the third is dense.
On exit, it is an array of dimension 4, with \textit{coltyp}(i) being the dimension of the \textit{i}-th type columns.

If \textit{info} = 0): successful exit

If \textit{info} = -i < 0, the \textit{i}-th argument had an illegal value.

\textbf{?lasd3}

\textit{Finds all square roots of the roots of the secular equation, as defined by the values in D and Z, and then updates the singular vectors by matrix multiplication. Used by ?bdsdc.}

\textbf{Syntax}

\begin{verbatim}
void slasd3( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, float *d, float *q, lapack_int *ldq, float *dsigma, float *u, lapack_int *ldu, float *u2, lapack_int *ldu2, float *vt, lapack_int *ldvt, float *vt2, lapack_int *ldvt2, lapack_int *idxc, lapack_int *ctot, float *z, lapack_int *info );
void dlasd3( lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, double *d, double *q, lapack_int *ldq, double *dsigma, double *u, lapack_int *ldu, double *u2, lapack_int *ldu2, double *vt, lapack_int *ldvt, double *vt2, lapack_int *ldvt2, lapack_int *idxc, lapack_int *ctot, double *z, lapack_int *info );
\end{verbatim}

\textbf{Include Files}

• mk1.h

\textbf{Description}

The routine \textit{?lasd3} finds all the square roots of the roots of the secular equation, as defined by the values in \textit{D} and \textit{Z}.

It makes the appropriate calls to \textit{?lasd4} and then updates the singular vectors by matrix multiplication.

The routine \textit{?lasd3} is called from \textit{?lasd1}.

\textbf{Input Parameters}

\begin{itemize}
  \item \textit{nl} \quad The row dimension of the upper block. \quad \textit{nl} \geq 1.
  \item \textit{nr} \quad The row dimension of the lower block. \quad \textit{nr} \geq 1.
  \item \textit{sqre} \quad If \textit{sqre} = 0): the lower block is an \textit{nr}-by-\textit{nr} square matrix.
  \item \textit{q} \quad Workspace array, \textit{DIMENSION} at least \textit{(ldq, k)}.
  \item \textit{ldq} \quad The leading dimension of the array \textit{Q}.
\end{itemize}
$ldq \geq k$.

$dsigma$

Array, DIMENSION $(k)$. The first $k$ elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

$ldu$

The leading dimension of the array $u$.

$ldu \geq n$.

$u2$

Array, DIMENSION $(ldu2, n)$.

The first $k$ columns of this matrix contain the non-deflated left singular vectors for the split problem.

$ldu2$

The leading dimension of the array $u2$.

$ldu2 \geq n$.

$ldvt$

The leading dimension of the array $vt$.

$ldvt \geq n$.

$vt2$

Array, DIMENSION $(ldvt2, n)$.

The first $k$ columns of $vt2'$ contain the non-deflated right singular vectors for the split problem.

$ldvt2$

The leading dimension of the array $vt2$.

$ldvt2 \geq n$.

$idxc$

Array, DIMENSION $(n)$.

The permutation used to arrange the columns of $u$ (and rows of $vt$) into three groups: the first group contains non-zero entries only at and above (or before) $nl +1$; the second contains non-zero entries only at and below (or after) $nl+2$; and the third is dense. The first column of $u$ and the row of $vt$ are treated separately, however. The rows of the singular vectors found by $?lasd4$ must be likewise permuted before the matrix multiplies can take place.

$ctot$

Array, DIMENSION $(4)$. A count of the total number of the various types of columns in $u$ (or rows in $vt$), as described in $idxc$.

The fourth column type is any column which has been deflated.

$z$

Array, DIMENSION $(k)$. The first $k$ elements of this array contain the components of the deflation-adjusted updating row vector.

**Output Parameters**

$d$

Array, DIMENSION $(k)$. On exit the square roots of the roots of the secular equation, in ascending order.

$u$

Array, DIMENSION $(ldu, n)$.

The last $n - k$ columns of this matrix contain the deflated left singular vectors.

$vt$

Array, DIMENSION $(ldvt, m)$.
The last \( m - k \) columns of \( vt' \) contain the deflated right singular vectors.

\[ vt2 \]

\[ z \]

\[ info \]

\( vt2 \) Destroyed on exit.

\( z \) Destroyed on exit.

If \( info = 0 \): successful exit.

If \( info = -i < 0 \), the \( i \)-th argument had an illegal value.

If \( info = 1 \), an singular value did not converge.

**Application Notes**

This code makes very mild assumptions about floating point arithmetic. It will work on machines with a guard digit in add/subtract, or on those binary machines without guard digits which subtract like the Cray XMP, Cray YMP, Cray C 90, or Cray 2. It could conceivably fail on hexadecimal or decimal machines without guard digits, but we know of none.

void slasd4( lapack_int *n, lapack_int *i, float *d, float *z, float *delta, float *rho, float *sigma, float *work, lapack_int *info);

void dlasd4( lapack_int *n, lapack_int *i, double *d, double *z, double *delta, double *rho, double *sigma, double *work, lapack_int *info);

**Include Files**

- mkl.h

**Description**

The routine computes the square root of the \( i \)-th updated eigenvalue of a positive symmetric rank-one modification to a positive diagonal matrix whose entries are given as the squares of the corresponding entries in the array \( d \), and that \( 0 \leq d(i) < d(j) \) for \( i < j \) and that \( \rho > 0 \). This is arranged by the calling routine, and is no loss in generality. The rank-one modified system is thus

\[
\text{diag}(d) \cdot \text{diag}(d) + \rho \cdot Z \cdot Z^T,
\]

where the Euclidean norm of \( Z \) is equal to 1. The method consists of approximating the rational functions in the secular equation by simpler interpolating rational functions.

**Input Parameters**

\[ n \]

The length of all arrays.

\[ i \]

The index of the eigenvalue to be computed. \( 1 \leq i \leq n \).

\[ d \]

Array, DIMENSION(\( n \)).
The original eigenvalues. They must be in order, \(0 \leq d(i) < d(j)\) for \(i < j\).

\(z\)
Array, \text{DIMENSION}\((n)\).
The components of the updating vector.

\(\rho\)
The scalar in the symmetric updating formula.

\(\text{work}\)
Workspace array, \text{DIMENSION}\((n)\).
If \(n\neq 1\), \text{work} contains \((d(j) + \sigma_i)\) in its \(j\)-th component.
If \(n = 1\), then \text{work}(1) = 1.

\textbf{Output Parameters}

\(\delta\)
Array, \text{DIMENSION}\((n)\).
If \(n\neq 1\), \(\delta\) contains \((d(j) - \sigma_i)\) in its \(j\)-th component.
If \(n = 1\), then \(\delta(1) = 1\). The vector \(\delta\) contains the information necessary to construct the (singular) eigenvectors.

\(\sigma\)
The computed \(\sigma_i\), the \(i\)-th updated eigenvalue.

\(\text{info}\)
\(= 0\): successful exit
\(> 0\): If \(\text{info} = 1\), the updating process failed.

\(\text{?lasd5}\)
\textit{Computes the square root of the } i\textit{-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix. Used by ?bdsdc.}\n
\textbf{Syntax}

\texttt{void slasd5( lapack_int *i, float *d, float *z, float *delta, float *rho, float *dsigma, float *work );}
\texttt{void dlasd5( lapack_int *i, double *d, double *z, double *delta, double *rho, double *dsigma, double *work );}

\textbf{Include Files}

• mkl.h

\textbf{Description}

The routine computes the square root of the \(i\)-th eigenvalue of a positive symmetric rank-one modification of a 2-by-2 diagonal matrix \(\text{diag}(d) \cdot \text{diag}(d) + \rho \cdot z \cdot z^T\).

The diagonal entries in the array \(d\) must satisfy \(0 \leq d(i) < d(j)\) for \(i < j\), \(\rho\) must be greater than 0, and that the Euclidean norm of the vector \(Z\) is equal to 1.

\textbf{Input Parameters}

\(i\)
The index of the eigenvalue to be computed. \(i = 1\) or \(i = 2\).
The original eigenvalues, $0 \leq d(1) < d(2)$.

The components of the updating vector.

The scalar in the symmetric updating formula.

Workspace array, dimension (2). Contains $(d(j) + sigma_i)$ in its $j$-th component.

Array, dimension (2).

Contains $(d(j) - sigma_i)$ in its $j$-th component. The vector $delta$ contains the information necessary to construct the eigenvectors.

The computed $sigma_i$, the $i$-th updated eigenvalue.

Computes the SVD of an updated upper bidiagonal matrix obtained by merging two smaller ones by appending a row. Used by ?bdsdc.

void slasd6( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre, float *d, float *vf, float *vl, float *alpha, float *beta, lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, float *givnum, lapack_int *ldgnum, float *poles, float *difl, float *difr, float *z, lapack_int *k, float *c, float *s, float *work, lapack_int *iwork, lapack_int *info);

void dlasd6( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre, double *d, double *vf, double *vl, double *alpha, double *beta, lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, double *givnum, lapack_int *ldgnum, double *poles, double *difl, double *difr, double *z, lapack_int *k, double *c, double *s, double *work, lapack_int *iwork, lapack_int *info);

Include Files

- mkl.h

Description

The routine ?lasd6 computes the SVD of an updated upper bidiagonal matrix $B$ obtained by merging two smaller ones by appending a row. This routine is used only for the problem which requires all singular values and optionally singular vector matrices in factored form. $B$ is an $n$-by-$m$ matrix with $n = nl + nr + 1$ and $m = n + sqre$. A related subroutine, ?lasd1, handles the case in which all singular values and singular vectors of the bidiagonal matrix are desired. ?lasd6 computes the SVD as follows:
\[
B = V^{(in)} D^{(in)} U^{(in)} \quad (Z^{(in)} = 0)
\]

where \(Z' = (Z1' \ aZ2' \ b) = u'^*VT'\), and \(u\) is a vector of dimension \(m\) with \(alpha\) and \(beta\) in the \(nl+1\) and \(nl+2\)-th entries and zeros elsewhere; and the entry \(b\) is empty if \(sqre = 0\).

The singular values of \(B\) can be computed using \(D1\), \(D2\), the first components of all the right singular vectors of the lower block, and the last components of all the right singular vectors of the upper block. These components are stored and updated in \(vf\) and \(vl\), respectively, in \(?lasd6\). Hence \(U\) and \(VT\) are not explicitly referenced.

The singular values are stored in \(D\). The algorithm consists of two stages:

1. The first stage consists of deflating the size of the problem when there are multiple singular values or if there is a zero in the \(Z\) vector. For each such occurrence the dimension of the secular equation problem is reduced by one. This stage is performed by the routine \(?lasd7\).
2. The second stage consists of calculating the updated singular values. This is done by finding the roots of the secular equation via the routine \(?lasd4\) (as called by \(?lasd8\)). This routine also updates \(vf\) and \(vl\) and computes the distances between the updated singular values and the old singular values. \(?lasd6\) is called from \(?lasda\).

**Input Parameters**

- **icompq**
  Specifies whether singular vectors are to be computed in factored form:
  - \(= 0\): Compute singular values only
  - \(= 1\): Compute singular vectors in factored form as well.

- **nl**
  The row dimension of the upper block.
  \(nl \geq 1\).

- **nr**
  The row dimension of the lower block.
  \(nr \geq 1\).

- **sqre**
  - \(= 0\): the lower block is an \(nr\)-by-\(nr\) square matrix.
  - \(= 1\): the lower block is an \(nr\)-by-(\(nr+1\)) rectangular matrix.

  The bidiagonal matrix has row dimension \(n=nl+nr+1\), and column dimension \(m = n + sqre\).

- **d**
  Array, dimension \((nl+nr+1)\). On entry \(d(1:nl,1:nl)\) contains the singular values of the upper block, and \(d(nl+2:n)\) contains the singular values of the lower block.

- **vf**
  Array, dimension \((m)\).
  On entry, \(vf(1:nl+1)\) contains the first components of all right singular vectors of the upper block; and \(vf(nl+2:m)\) contains the first components of all right singular vectors of the lower block.

- **vl**
  Array, dimension \((m)\).
On entry, \( vl(1:n_l+1) \) contains the last components of all right singular vectors of the upper block; and \( vl(n_l+2:m) \) contains the last components of all right singular vectors of the lower block.

- **alpha**: Contains the diagonal element associated with the added row.
- **beta**: Contains the off-diagonal element associated with the added row.
- **ldgcol**: The leading dimension of the output array \( givcol \), must be at least \( n \).
- **ldgnum**: The leading dimension of the output arrays \( givnum \) and \( poles \), must be at least \( n \).
- **work**: Workspace array, dimension \( (4m) \).
- **iwork**: Workspace array, dimension \( (3n) \).

**Output Parameters**

- **d**: On exit \( d(1:n) \) contains the singular values of the modified matrix.
- **vf**: On exit, \( vf \) contains the first components of all right singular vectors of the bidiagonal matrix.
- **vl**: On exit, \( vl \) contains the last components of all right singular vectors of the bidiagonal matrix.
- **alpha**: On exit, the diagonal element associated with the added row deflated by \( \max(\|\alpha\|, \|\beta\|, \|D(I)\|), I = 1,n \).
- **beta**: On exit, the off-diagonal element associated with the added row deflated by \( \max(\|\alpha\|, \|\beta\|, \|D(I)\|), I = 1,n \).
- **idxq**: Array, dimension \( (n) \). This contains the permutation which will reintegrate the subproblem just solved back into sorted order, that is, \( d( idxq(i = 1, n) ) \) will be in ascending order.
- **perm**: Array, dimension \( (n) \). The permutations (from deflation and sorting) to be applied to each block. Not referenced if \( icompq = 0 \).
- **givpstr**: The number of Givens rotations which took place in this subproblem. Not referenced if \( icompq = 0 \).
- **givcol**: Array, dimension \( (ldgcol, 2) \). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if \( icompq = 0 \).
- **givnum**: Array, dimension \( (ldgnum, 2) \). Each number indicates the C or S value to be used in the corresponding Givens rotation. Not referenced if \( icompq = 0 \).
- **poles**: Array, dimension \( (ldgnum, 2) \). On exit, \( poles(1,*) \) is an array containing the new singular values obtained from solving the secular equation, and \( poles(2,*) \) is an array containing the poles in the secular equation. Not referenced if \( icompq = 0 \).
- **difl**: Array, dimension \( (n) \). On exit, \( difl(i) \) is the distance between \( i \)-th updated (undeflated) singular value and the \( i \)-th (undeflated) old singular value.
difr: Array, dimension (ldgnum, 2) if icompq = 1 and dimension (n) if icompq = 0.

On exit, difr(i, 1) is the distance between i-th updated (undeflated) singular value and the i+1-th (undeflated) old singular value. If icompq = 1, difr(1: k, 2) is an array containing the normalizing factors for the right singular vector matrix.

See ?lasd8 for details on difl and difr.

z: Array, dimension (m).

The first elements of this array contain the components of the deflation-adjusted updating row vector.

k: Contains the dimension of the non-deflated matrix. This is the order of the related secular equation. 1 ≤ k ≤ n.

c: c contains garbage if sqre = 0 and the C-value of a Givens rotation related to the right null space if sqre = 1.

s: s contains garbage if sqre = 0 and the S-value of a Givens rotation related to the right null space if sqre = 1.

info: info = 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.
> 0: if info = 1, an singular value did not converge

?lasd7: Merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. Used by ?bdsdc.

Syntax
void slasd7( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, float *d, float *z, float *zw, float *vf, float *vfw, float *vl, float *vlw, float *alpha, float *beta, float *dsigma, lapack_int *idx, lapack_int *idxp, lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, float *givnum, lapack_int *ldgnum, float *c, float *s, lapack_int *info );
void dlasd7( lapack_int *icompq, lapack_int *nl, lapack_int *nr, lapack_int *sqre, lapack_int *k, double *d, double *z, double *zw, double *vf, double *vfw, double *vl, double *vlw, double *alpha, double *beta, double *dsigma, lapack_int *idx, lapack_int *idxp, lapack_int *idxq, lapack_int *perm, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, double *givnum, lapack_int *ldgnum, double *c, double *s, lapack_int *info );

Include Files
• mkl.h

Description
The routine `?lasd7` merges the two sets of singular values together into a single sorted set. Then it tries to deflate the size of the problem. There are two ways in which deflation can occur: when two or more singular values are close together or if there is a tiny entry in the Z vector. For each such occurrence the order of the related secular equation problem is reduced by one. `?lasd7` is called from `?lasd6`.

**Input Parameters**

- **icompq**
  Specifies whether singular vectors are to be computed in compact form, as follows:
  - = 0: Compute singular values only.
  - = 1: Compute singular vectors of upper bidiagonal matrix in compact form.

- **nl**
  The row dimension of the upper block.
  \( nl \geq 1 \).

- **nr**
  The row dimension of the lower block.
  \( nr \geq 1 \).

- **sqre**
  - = 0: the lower block is an \( nr \)-by-\( nr \) square matrix.
  - = 1: the lower block is an \( nr \)-by-(\( nr+1 \)) rectangular matrix. The bidiagonal matrix has \( n = nl + nr + 1 \) rows and \( m = n + sqre \) columns.

- **d**
  Array, DIMENSION (\( n \)). On entry \( d \) contains the singular values of the two submatrices to be combined.

- **zw**
  Array, DIMENSION (\( m \)).
  Workspace for \( z \).

- **vf**
  Array, DIMENSION (\( m \)). On entry, \( vf(1:nl+1) \) contains the first components of all right singular vectors of the upper block; and \( vf(nl+2:m) \) contains the first components of all right singular vectors of the lower block.

- **vfw**
  Array, DIMENSION (\( m \)).
  Workspace for \( vf \).

- **vl**
  Array, DIMENSION (\( m \)).
  On entry, \( vl(1:nl+1) \) contains the last components of all right singular vectors of the upper block; and \( vl(nl+2:m) \) contains the last components of all right singular vectors of the lower block.

- **VLW**
  Array, DIMENSION (\( m \)).
  Workspace for \( VL \).

- **alpha**
  REAL for `slasd7`
  DOUBLE PRECISION for `dlasd7`.
  Contains the diagonal element associated with the added row.

- **beta**
  Contains the off-diagonal element associated with the added row.

- **idx**
  Workspace array, DIMENSION (\( n \)). This will contain the permutation used to sort the contents of \( d \) into ascending order.
Workspace array, DIMENSION (n). This will contain the permutation used to place deflated values of \( d \) at the end of the array.

Array, DIMENSION (n).

This contains the permutation which separately sorts the two sub-problems in \( d \) into ascending order. Note that entries in the first half of this permutation must first be moved one position backward; and entries in the second half must first have \( nl+1 \) added to their values.

The leading dimension of the output array \( givcol \), must be at least \( n \).

The leading dimension of the output array \( givnum \), must be at least \( n \).

**Output Parameters**

Contains the dimension of the non-deflated matrix, this is the order of the related secular equation.

\( 1 \leq k \leq n \).

On exit, \( d \) contains the trailing \( (n-k) \) updated singular values (those which were deflated) sorted into increasing order.

Array, DIMENSION (\( m \)).

On exit, \( Z \) contains the updating row vector in the secular equation.

On exit, \( vf \) contains the first components of all right singular vectors of the bidiagonal matrix.

On exit, \( vl \) contains the last components of all right singular vectors of the bidiagonal matrix.

Array, DIMENSION (n). Contains a copy of the diagonal elements \( (k-1) \) singular values and one zero) in the secular equation.

On output, \( idxp(2: k) \) points to the nondeflated \( d \)-values and \( idxp( k+1:n) \) points to the deflated singular values.

Array, DIMENSION (n).

The permutations (from deflation and sorting) to be applied to each singular block. Not referenced if \( icompq = 0 \).

The number of Givens rotations which took place in this subproblem. Not referenced if \( icompq = 0 \).

Array, DIMENSION ( \( ldgcol \), 2 ). Each pair of numbers indicates a pair of columns to take place in a Givens rotation. Not referenced if \( icompq = 0 \).

Array, DIMENSION ( \( ldgnum \), 2 ). Each number indicates the \( C \) or \( S \) value to be used in the corresponding Givens rotation. Not referenced if \( icompq = 0 \).

If \( sqre = 0 \), then \( c \) contains garbage, and if \( sqre = 1 \), then \( c \) contains \( C \)-value of a Givens rotation related to the right null space.

If \( sqre = 0 \), then \( s \) contains garbage, and if \( sqre = 1 \), then \( s \) contains \( S \)-value of a Givens rotation related to the right null space.
info

= 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.

?lasd8

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

void slasd8 ( lapack_int *icompq, lapack_int *k, float *d, float *z, float *vf, float *vl, float *difl, float *difr, lapack_int *lddifr, float *dsigma, float *work,
  lapack_int *info );

void dlasd8 ( lapack_int *icompq, lapack_int *k, double *d, double *z, double *vf, double *vl, double *difl, double *difr, lapack_int *lddifr, double *dsigma, double *work,
  lapack_int *info );

Include Files

• mkl.h

Description

The routine ?lasd8 finds the square roots of the roots of the secular equation, as defined by the values in dsigma and z. It makes the appropriate calls to ?lasd4, and stores, for each element in d, the distance to its two nearest poles (elements in dsigma). It also updates the arrays vf and vl, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd8 is called from ?lasd6.

Input Parameters

icompq

Specifies whether singular vectors are to be computed in factored form in the calling routine:

= 0: Compute singular values only.
= 1: Compute singular vectors in factored form as well.

k

The number of terms in the rational function to be solved by ?lasd4. k ≥ 1.

z

Array, DIMENSION ( k ).

The first k elements of this array contain the components of the deflation-adjusted updating row vector.

vf

Array, DIMENSION ( k ).

On entry, vf contains information passed through dbede8.

vl

Array, DIMENSION ( k ). On entry, vl contains information passed through dbede8.

lddifr

The leading dimension of the output array difr, must be at least k.

dsigma

Array, DIMENSION ( k ).

The first k elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.
work

Workspace array, DIMENSION at least (3k).

Output Parameters

d
Array, DIMENSION ( k ).
On output, D contains the updated singular values.

z
Updated on exit.

vf
On exit, vf contains the first k components of the first components of all right singular vectors of the bidiagonal matrix.

vl
On exit, vl contains the first k components of the last components of all right singular vectors of the bidiagonal matrix.

difl
Array, DIMENSION ( k ). On exit, difl(i) = d(i) - dsigma(i).

difr
Array,
DIMENSION ( lddifr, 2 ) if icompq = 1 and
DIMENSION ( k ) if icompq = 0.
On exit, difr(i,1) = d(i) - dsigma(i+1), difr(k,1) is not defined and will not be referenced. If icompq = 1, difr(1:k,2) is an array containing the normalizing factors for the right singular vector matrix.

dsigma
The elements of this array may be very slightly altered in value.

info
= 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.
> 0: If info = 1, an singular value did not converge.

?lasd9

Finds the square roots of the roots of the secular equation, and stores, for each element in D, the distance to its two nearest poles. Used by ?bdsdc.

Syntax

void slasd9( lapack_int *icompq, lapack_int *k, float *d, float *z, float *vf, float *vl, float *difl, float *difr, float *dsigma, float *work, lapack_int *info );
void dlasd9( lapack_int *icompq, lapack_int *k, double *d, double *z, double *vf, double *vl, double *difl, double *difr, double *dsigma, double *work, lapack_int *info );

Include Files

• mkl.h

Description

The routine ?lasd9 finds the square roots of the roots of the secular equation, as defined by the values in dsigma and z. It makes the appropriate calls to ?lasd4, and stores, for each element in d, the distance to its two nearest poles (elements in dsigma). It also updates the arrays vf and vl, the first and last components of all the right singular vectors of the original bidiagonal matrix. ?lasd9 is called from ?lasd7.
Input Parameters

icomq

Specifies whether singular vectors are to be computed in factored form in the calling routine:

If icomq = 0, compute singular values only;
If icomq = 1, compute singular vector matrices in factored form also.

k

The number of terms in the rational function to be solved by slasd4. k ≥ 1.

dsigma

Array, DIMENSION(k).
The first k elements of this array contain the old roots of the deflated updating problem. These are the poles of the secular equation.

z

Array, DIMENSION(k). The first k elements of this array contain the components of the deflation-adjusted updating row vector.

vf

Array, DIMENSION(k). On entry, vf contains information passed through sbede8.

vl

Array, DIMENSION(k). On entry, vl contains information passed through sbede8.

work

Workspace array, DIMENSION at least (3k).

Output Parameters

d

Array, DIMENSION(k). d(i) contains the updated singular values.

vf

On exit, vf contains the first k components of the first components of all right singular vectors of the bidiagonal matrix.

vl

On exit, vl contains the first k components of the last components of all right singular vectors of the bidiagonal matrix.

difl

Array, DIMENSION(k).
On exit, difl(i) = d(i) - dsigma(i).

difr

Array,
DIMENSION (ldu, 2) if icomq = 1 and
DIMENSION (k) if icomq = 0.
On exit, difr(i, 1) = d(i) - dsigma(i+1), difr(k, 1) is not defined and will not be referenced.
If icomq = 1, difr(1:k, 2) is an array containing the normalizing factors for the right singular vector matrix.

info

= 0: successful exit.
< 0: if info = -i, the i-th argument had an illegal value.
> 0: If info = 1, an singular value did not converge
?lasda

Computes the singular value decomposition (SVD) of a real upper bidiagonal matrix with diagonal d and off-diagonal e. Used by ?bdsdc.

Syntax

void slasda( lapack_int *icompq, lapack_int *smlsiz, lapack_int *n, lapack_int *sqre, float *d, float *e, float *u, lapack_int *ldu, float *vt, lapack_int *k, float *difl, float *difr, float *z, float *poles, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, lapack_int *perm, float *givnum, float *c, float *s, float *work, lapack_int *iwork, lapack_int *info );

void dlasda( lapack_int *icompq, lapack_int *smlsiz, lapack_int *n, lapack_int *sqre, double *d, double *e, double *u, lapack_int *ldu, double *vt, lapack_int *k, double *difl, double *difr, double *z, double *poles, lapack_int *givptr, lapack_int *givcol, lapack_int *ldgcol, lapack_int *perm, double *givnum, double *c, double *s, double *work, lapack_int *iwork, lapack_int *info );

Include Files

• mkl.h

Description

Using a divide and conquer approach, ?lasda computes the singular value decomposition (SVD) of a real upper bidiagonal n-by-m matrix B with diagonal d and off-diagonal e, where m = n + sqre.

The algorithm computes the singular values in the $\text{SVDB} = U*S*VT$. The orthogonal matrices $U$ and $VT$ are optionally computed in compact form. A related subroutine ?lasd0 computes the singular values and the singular vectors in explicit form.

Input Parameters

icompq

Specifies whether singular vectors are to be computed in compact form, as follows:

= 0: Compute singular values only.

= 1: Compute singular vectors of upper bidiagonal matrix in compact form.

smlsiz

The maximum size of the subproblems at the bottom of the computation tree.

n

The row dimension of the upper bidiagonal matrix. This is also the dimension of the main diagonal array d.

sqre

Specifies the column dimension of the bidiagonal matrix.

If sqre = 0: the bidiagonal matrix has column dimension $m = n$

If sqre = 1: the bidiagonal matrix has column dimension $m = n + 1$.

d

Array, DIMENSION (n). On entry, d contains the main diagonal of the bidiagonal matrix.

e

Array, DIMENSION (m - 1). Contains the subdiagonal entries of the bidiagonal matrix. On exit, e is destroyed.
The leading dimension of arrays $u$, $vt$, $difl$, $difr$, $poles$, $givnum$, and $z$. $ldu \geq n$.

The leading dimension of arrays $givcol$ and $perm$. $ldgcol \geq n$.

Workspace array, DIMENSION $(6n+smlsiz+1)^2$.

Workspace array, Dimension must be at least $(7n)$.

**Output Parameters**

On exit $d$, if $info = 0$, contains the singular values of the bidiagonal matrix.

Array, DIMENSION $(ldu, smlsiz)$ if $icompq = 1$.

Not referenced if $icompq = 0$.

If $icompq = 1$, on exit, $u$ contains the left singular vector matrices of all subproblems at the bottom level.

Array, DIMENSION $(ldu, smlsiz+1)$ if $icompq = 1$, and not referenced if $icompq = 0$. If $icompq = 1$, on exit, $vt'$ contains the right singular vector matrices of all subproblems at the bottom level.

Array, DIMENSION $(n)$ if $icompq = 1$ and

DIMENSION $(1)$ if $icompq = 0$.

If $icompq = 1$, on exit, $k(i)$ is the dimension of the $i$-th secular equation on the computation tree.

REAL for slasda

DOUBLE PRECISION for dlasda.

Array, DIMENSION $(ldu, nlvl)$,

where $nlvl = \text{floor}(\log_2(n/smlsiz))$.

Array,

DIMENSION $(ldu, 2nlvl)$ if $icompq = 1$ and

DIMENSION $(n)$ if $icompq = 0$.

If $icompq = 1$, on exit, $difl(1:n, i)$ and $difr(1:n, 2i-1)$ record distances between singular values on the $i$-th level and singular values on the $(i-1)$-th level, and $difr(1:n, 2i)$ contains the normalizing factors for the right singular vector matrix. See \texttt{?lasd8} for details.

Array,

DIMENSION $(ldu, nlvl)$ if $icompq = 1$ and

DIMENSION $(n)$ if $icompq = 0$. The first $k$ elements of $z(1, i)$ contain the components of the deflation-adjusted updating row vector for subproblems on the $i$-th level.

Array, DIMENSION$(ldu, 2*nlvl)$
if $\text{icompq} = 1$, and not referenced if $\text{icompq} = 0$. If $\text{icompq} = 1$, on exit, $\text{poles}(1, 2i - 1)$ and $\text{poles}(1, 2i)$ contain the new and old singular values involved in the secular equations on the $i$-th level.

$\text{givptr}$ Array, DIMENSION $(n)$ if $\text{icompq} = 1$, and not referenced if $\text{icompq} = 0$. If $\text{icompq} = 1$, on exit, $\text{givptr}(i)$ records the number of Givens rotations performed on the $i$-th problem on the computation tree.

$\text{givcol}$ Array, DIMENSION $(\text{ldgcol}, 2*\text{nlvl})$ if $\text{icompq} = 1$, and not referenced if $\text{icompq} = 0$. If $\text{icompq} = 1$, on exit, for each $i$, $\text{givcol}(1, 2i - 1)$ and $\text{givcol}(1, 2i)$ record the locations of Givens rotations performed on the $i$-th level on the computation tree.

$\text{perm}$ Array, DIMENSION $(\text{ldgcol}, 2*\text{nlvl})$ if $\text{icompq} = 1$, and not referenced if $\text{icompq} = 0$. If $\text{icompq} = 1$, on exit, $\text{perm}(1, i)$ records permutations done on the $i$-th level of the computation tree.

$\text{givnum}$ Array, DIMENSION $(\text{ldu}, 2*\text{nlvl})$ if $\text{icompq} = 1$, and not referenced if $\text{icompq} = 0$. If $\text{icompq} = 1$, on exit, for each $i$, $\text{givnum}(1, 2i - 1)$ and $\text{givnum}(1, 2i)$ record the $C$- and $S$-values of Givens rotations performed on the $i$-th level on the computation tree.

$c$ Array,
DIMENSION $(n)$ if $\text{icompq} = 1$, and
DIMENSION $(1)$ if $\text{icompq} = 0$.
If $\text{icompq} = 1$ and the $i$-th subproblem is not square, on exit, $c(i)$ contains the $C$-value of a Givens rotation related to the right null space of the $i$-th subproblem.

$s$ Array,
DIMENSION $(n)$ if $\text{icompq} = 1$, and
DIMENSION $(1)$ if $\text{icompq} = 0$.
If $\text{icompq} = 1$ and the $i$-th subproblem is not square, on exit, $s(i)$ contains the $S$-value of a Givens rotation related to the right null space of the $i$-th subproblem.

$info$ = 0: successful exit.
< 0: if $info = -i$, the $i$-th argument had an illegal value
> 0: If $info = 1$, an singular value did not converge

$\text{slasdq}$
Computes the SVD of a real bidiagonal matrix with diagonal $d$ and off-diagonal $e$. Used by $\text{bdsdc}$.

Syntax
void slasdq( char *uplo, lapack_int *sqre, lapack_int *n, lapack_int *ncvt, lapack_int *nru, lapack_int *ncc, float *d, float *e, float *vt, lapack_int *ldvt, float *u, lapack_int *ldu, float *c, lapack_int *ldc, float *work, lapack_int *info );
void dlasdq( char *uplo, lapack_int *sqre, lapack_int *n, lapack_int *ncvt, lapack_int *nru, lapack_int *ncc, double *d, double *e, double *vt, lapack_int *ldvt, double *u, lapack_int *ldu, double *c, lapack_int *ldc, double *work, lapack_int *info );

Include Files

- mkl.h

Description

The routine ?lasdq computes the singular value decomposition (SVD) of a real (upper or lower) bidiagonal matrix with diagonal \( d \) and off-diagonal \( e \), accumulating the transformations if desired. If \( B \) is the input bidiagonal matrix, the algorithm computes orthogonal matrices \( Q \) and \( P \) such that \( B = Q \cdot S \cdot P^T \). The singular values \( S \) are overwritten on \( d \).

The input matrix \( U \) is changed to \( U \cdot Q \) if desired.

The input matrix \( VT \) is changed to \( P^T \cdot VT \) if desired.

The input matrix \( C \) is changed to \( Q^T \cdot C \) if desired.

Input Parameters

\textbf{uplo}

On entry, \textit{uplo} specifies whether the input bidiagonal matrix is upper or lower bidiagonal.

If \textit{uplo} = 'U' or 'u', \( B \) is upper bidiagonal;

If \textit{uplo} = 'L' or 'l', \( B \) is lower bidiagonal.

\textbf{sqre}

\( = 0 \): then the input matrix is \( n \)-by-\( n \).

\( = 1 \): then the input matrix is \( n \)-by-(\( n + 1 \)) if \textit{uplo} = 'U' and \( (n + 1) \)-by-\( n \) if \textit{uplo} = 'L'. The bidiagonal matrix has \( n = n \cdot l + n \cdot r + 1 \) rows and \( m = n + sqre \geq n \) columns.

\textbf{n}

On entry, \( n \) specifies the number of rows and columns in the matrix. \( n \) must be at least 0.

\textbf{ncvt}

On entry, \( ncvt \) specifies the number of columns of the matrix \( VT \). \( ncvt \) must be at least 0.

\textbf{nru}

On entry, \( nru \) specifies the number of rows of the matrix \( U \). \( nru \) must be at least 0.

\textbf{ncc}

On entry, \( ncc \) specifies the number of columns of the matrix \( C \). \( ncc \) must be at least 0.

\textbf{d}

Array, DIMENSION \( (n) \). On entry, \( d \) contains the diagonal entries of the bidiagonal matrix.

\textbf{e}

Array, DIMENSION is \( (n-1) \) if \( sqre = 0 \) and \( n \) if \( sqre = 1 \). On entry, the entries of \( e \) contain the off-diagonal entries of the bidiagonal matrix.

\textbf{vt}

Array, DIMENSION \( (ldvt, ncvt) \). On entry, contains a matrix which on exit has been premultiplied by \( P^T \), dimension \( n \)-by-\( ncvt \) if \( sqre = 0 \) and \( (n + 1) \)-by-\( ncvt \) if \( sqre = 1 \) (not referenced if \( ncvt = 0 \)).
On entry, \textit{ldvt} specifies the leading dimension of \textit{vt} as declared in the calling (sub) program. \textit{ldvt} must be at least 1. If \textit{ncvt} is nonzero, \textit{ldvt} must also be at least \textit{n}.

\textit{u} Array, \textit{DIMENSION} (\textit{ldu}, \textit{n}). On entry, contains a matrix which on exit has been postmultiplied by \textit{Q}, dimension \textit{nru}-by-\textit{n} if \textit{sqre} = 0 and \textit{nru}-by-(\textit{n} + 1) if \textit{sqre} = 1 (not referenced if \textit{nru}=0).

On entry, \textit{ldu} specifies the leading dimension of \textit{u} as declared in the calling (sub) program. \textit{ldu} must be at least \textit{max}(1, \textit{nru}).

\textit{c} Array, \textit{DIMENSION} (\textit{ldc}, \textit{ncc}). On entry, contains an \textit{n}-by-\textit{ncc} matrix which on exit has been premultiplied by \textit{Q'}, dimension \textit{n}-by-\textit{ncc} if \textit{sqre} = 0 and (\textit{n}+1)-by-\textit{ncc} if \textit{sqre} = 1 (not referenced if \textit{ncc}=0).

On entry, \textit{ldc} specifies the leading dimension of \textit{C} as declared in the calling (sub) program. \textit{ldc} must be at least 1. If \textit{ncc} is non-zero, \textit{ldc} must also be at least \textit{n}.

\textit{work} Array, \textit{DIMENSION} (4\textit{n}). This is a workspace array. Only referenced if one of \textit{ncvt}, \textit{nru}, or \textit{ncc} is nonzero, and if \textit{n} is at least 2.

**Output Parameters**

\textit{d} On normal exit, \textit{d} contains the singular values in ascending order.

\textit{e} On normal exit, \textit{e} will contain 0. If the algorithm does not converge, \textit{d} and \textit{e} will contain the diagonal and superdiagonal entries of a bidiagonal matrix orthogonally equivalent to the one given as input.

On exit, the matrix has been premultiplied by \textit{P}'.

On exit, the matrix has been postmultiplied by \textit{Q}.

On exit, the matrix has been premultiplied by \textit{Q'}.

On exit, a value of 0 indicates a successful exit. If \textit{info} < 0, argument number \textit{-info} is illegal. If \textit{info} > 0, the algorithm did not converge, and \textit{info} specifies how many superdiagonals did not converge.

**?lasdt**

*Creates a tree of subproblems for bidiagonal divide and conquer. Used by ?bdsdc.*

**Syntax**

```c
void slasdt( lapack_int *n, lapack_int *lvl, lapack_int *nd, lapack_int *inode,
            lapack_int *ndiml, lapack_int *ndimr, lapack_int *msub );
void dlasdt( lapack_int *n, lapack_int *lvl, lapack_int *nd, lapack_int *inode,
            lapack_int *ndiml, lapack_int *ndimr, lapack_int *msub );
```

**Include Files**

- mkl.h
Description

The routine creates a tree of subproblems for bidiagonal divide and conquer.

Input Parameters

- \( n \)
  - On entry, the number of diagonal elements of the bidiagonal matrix.
- \( m_{sub} \)
  - On entry, the maximum row dimension each subproblem at the bottom of the tree can be of.

Output Parameters

- \( lvl \)
  - On exit, the number of levels on the computation tree.
- \( nd \)
  - On exit, the number of nodes on the tree.
- \( inode \)
  - Array, \( \text{DIMENSION} (n) \). On exit, centers of subproblems.
- \( ndiml \)
  - Array, \( \text{DIMENSION} (n) \). On exit, row dimensions of left children.
- \( ndimr \)
  - Array, \( \text{DIMENSION} (n) \). On exit, row dimensions of right children.

?laset

Initializes the off-diagonal elements and the diagonal elements of a matrix to given values.

Syntax

```c
lapack_int LAPACKE_slaset (int matrix_layout, char uplo, lapack_int m, lapack_int n, float alpha, float beta, float * a, lapack_int lda);
```

```c
lapack_int LAPACKE_dlaset (int matrix_layout, char uplo, lapack_int m, lapack_int n, double alpha, double beta, double * a, lapack_int lda);
```

```c
lapack_int LAPACKE_claset (int matrix_layout, char uplo, lapack_int m, lapack_int n, lapack_complex_float alpha, lapack_complex_float beta, lapack_complex_float * a, lapack_int lda);
```

```c
lapack_int LAPACKE_zlaset (int matrix_layout, char uplo, lapack_int m, lapack_int n, lapack_complex_double alpha, lapack_complex_double beta, lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine initializes an \( m \)-by-\( n \) matrix \( A \) to \( beta \) on the diagonal and \( alpha \) on the off-diagonals.

Input Parameters

- \( matrix\_layout \)
  - Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- \( uplo \)
  - Specifies the part of the matrix \( A \) to be set.
If $\text{uplo} = 'U'$, upper triangular part is set; the strictly lower triangular part of $A$ is not changed.

If $\text{uplo} = 'L'$, lower triangular part is set; the strictly upper triangular part of $A$ is not changed.

Otherwise: All of the matrix $A$ is set.

$m$
The number of rows of the matrix $A$. $m \geq 0$.

$n$
The number of columns of the matrix $A$. $n \geq 0$.

$\alpha, \beta$
The constants to which the off-diagonal and diagonal elements are to be set, respectively.

$a$
Array, size at least $\max(1, \text{lda} \times n)$ for column major and $\max(1, \text{lda} \times m)$ for row major layout.

The array $a$ contains the $m$-by-$n$ matrix $A$.

$\text{lda}$
The leading dimension of the array $a$.

$\text{lda} \geq \max(1, m)$ for column major layout and $\text{lda} \geq \max(1, n)$ for row major layout.

Output Parameters

On exit, the leading $m$-by-$n$ submatrix of $A$ is set as follows:

- If $\text{uplo} = 'U'$, $A_{ij} = \alpha$, $1 \leq i \leq j - 1$, $1 \leq j \leq n$,
- If $\text{uplo} = 'L'$, $A_{ij} = \alpha$, $j + 1 \leq i \leq m$, $1 \leq j \leq n$,
- otherwise, $A_{ij} = \alpha$, $1 \leq i \leq m$, $1 \leq j \leq n$, $i \neq j$,
- and, for all $\text{uplo}$, $A_{ij} = \beta$, $1 \leq i \leq \min(m, n)$.

Return Values

This function returns a value $\text{info}$.

- If $\text{info} = 0$, the execution is successful.
- If $\text{info} = i < 0$, the $i$-th parameter had an illegal value.
- If $\text{info} = -1011$, memory allocation error occurred.

?lasrt

Sorts numbers in increasing or decreasing order.

Syntax

```c
lapack_int LAPACKE_slasrt (char id , lapack_int n , float * d );
lapack_int LAPACKE_dlasrt (char id , lapack_int n , double * d );
```

Include Files

- `mkl.h`
Description

The routine `?lasrt` sorts the numbers in `d` in increasing order (if `id = 'I'`) or in decreasing order (if `id = 'D'`). It uses Quick Sort, reverting to Insertion Sort on arrays of size \( \leq 20 \). Dimension of stack limits \( n \) to about \( 2^{32} \).

Input Parameters

\[ id \]
- `id = 'I'`: sort `d` in increasing order;
- `id = 'D'`: sort `d` in decreasing order.

\[ n \]
The length of the array `d`.

\[ d \]
On entry, the array to be sorted.

Output Parameters

\[ d \]
On exit, `d` has been sorted into increasing order

\[ d[0] \leq d[1] \leq ... \leq d[n-1] \] or into decreasing order

\[ d[0] \geq d[1] \geq ... \geq d[n-1] \], depending on `id`.

Return Values

This function returns a value `info`.

If `info = 0`, the execution is successful.

If `info < 0`, the i-th parameter had an illegal value.

?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

```c
lapack_int LAPACKE_slaswp (int matrix_layout, lapack_int n, float * a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int * ipiv, lapack_int incx);
```

```c
lapack_int LAPACKE_dlaswp (int matrix_layout, lapack_int n, double * a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int * ipiv, lapack_int incx);
```

```c
lapack_int LAPACKE_claswp (int matrix_layout, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int * ipiv, lapack_int incx);
```

```c
lapack_int LAPACKE_zlaswp (int matrix_layout, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_int k1, lapack_int k2, const lapack_int * ipiv, lapack_int incx);
```

Include Files

- mkl.h

Description

The routine performs a series of row interchanges on the matrix `A`. One row interchange is initiated for each of rows `k1` through `k2` of `A`. 
Input Parameters

matrix_layout
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

n
  The number of columns of the matrix A.

a
  Array, size max(1, lda*n) for column major and max(1, lda*mm) for row major layout. Here mm is not less than maximum of values ipiv[k1-1+j*|incx|], 0≤j<k2-k1.

Array a contains the m-by-n matrix A.

lda
  The leading dimension of the array a.

k1
  The first element of ipiv for which a row interchange will be done.

k2
  The last element of ipiv for which a row interchange will be done.

ipiv
  Array, size k1+(k2-k1)*|incx|).

The vector of pivot indices. Only the elements in positions k1 through k2 of ipiv are accessed.

ipiv(k) = l implies rows k and l are to be interchanged.

incx
  The increment between successive values of ipiv. If ipiv is negative, the pivots are applied in reverse order.

Output Parameters

a
  On exit, the permuted matrix.

Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info = -i, the i-th parameter had an illegal value.

If info = -1011, memory allocation error occurred.

?latm1

Computes the entries of a matrix as specified.

Syntax

void slatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, float *d, lapack_int *n, lapack_int *info);
void dlatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, double *d, lapack_int *n, lapack_int *info);
void clatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, lapack_complex *d, lapack_int *n, lapack_int *info);
void zlatm1 (lapack_int *mode, *cond, lapack_int *irsign, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *n, lapack_int *info);
Include Files

- mkl.h

Description

The ?latm1 routine computes the entries of \(D(1..n)\) as specified by mode, cond and irsign. idist and iseed determine the generation of random numbers.

?latm1 is called by slatmr (for slatm1 and dlatm1), and by clatmr (for clatm1 and zlatm1) to generate random test matrices for LAPACK programs.

Input Parameters

- **mode**
  
  On entry describes how \(d\) is to be computed:
  
  - \(mode = 0\) means do not change \(d\).
  - \(mode = 1\) sets \(d[0] = 1\) and \(d[1:n - 1] = 1.0/\text{cond}\)
  - \(mode = 2\) sets \(d[0:n - 2] = 1\) and \(d[n - 1] = 1.0/\text{cond}\)
  - \(mode = 3\) sets \(d[i - 1] = \text{cond}**(-(i-1)/(n-1))\)
  - \(mode = 4\) sets \(d[i - 1] = 1 - (i-1)/(n-1)*(1 - 1/\text{cond})\)
  - \(mode = 5\) sets \(d\) to random numbers in the range \((1/\text{cond}, 1)\) such that their logarithms are uniformly distributed.
  - \(mode = 6\) sets \(d\) to random numbers from same distribution as the rest of the matrix.
  
  \(mode < 0\) has the same meaning as abs(\(mode\)), except that the order of the elements of \(d\) is reversed.
  
  Thus if \(mode\) is positive, \(d\) has entries ranging from 1 to 1/\text{cond}, if negative, from 1/\text{cond} to 1.

- **cond**
  
  On entry, used as described under mode above. If used, it must be \(\geq 1\).

- **irsign**
  
  On entry, if \(mode\) is not -6, 0, or 6, determines sign of entries of \(d\).
  
  If \(irsign = 0\), entries of \(d\) are unchanged.
  
  If \(irsign = 1\), each entry of \(d\) is multiplied by a random complex number uniformly distributed with absolute value 1.

- **idist**
  
  Specifies the distribution of the random numbers.
  
  For slatm1 and dlatm1:
  
  - \(= 1\): uniform \((0,1)\)
  - \(= 2\): uniform \((-1,1)\)
  - \(= 3\): normal \((0,1)\)
  
  For clatm1 and zlatm1:
  
  - \(= 1\): real and imaginary parts each uniform \((0,1)\)
  - \(= 2\): real and imaginary parts each uniform \((-1,1)\)
  - \(= 3\): real and imaginary parts each normal \((0,1)\)
  - \(= 4\): complex number uniform in disk(0, 1)
iseed Array, size (4).

Specifies the seed of the random number generator. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of iseed[3] are changed on exit, and can be used in the next call to ?latm1 to continue the same random number sequence.

d Array, size n.

Number of entries of d.

Output Parameters

iseed On exit, the seed is updated.

If info = 0, the execution is successful.

If info = -1, mode is not in range -6 to 6.

If info = -2, mode is neither -6, 0 nor 6, and irsign is neither 0 nor 1.

If info = -3, mode is neither -6, 0 nor 6 and cond is less than 1.

If info = -4, mode equals 6 or -6 and idist is not in range 1 to 4.

If info = -7, n is negative.

?latm2

Returns an entry of a random matrix.

Syntax

float slatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, float *d, lapack_int *igrade, float *dl, float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

double dlatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, double *d, lapack_int *igrade, double *dl, double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

For non-gfortran (libmkl_intel_*) interface libraries:

void clatm2 (lapack_complex_float *res, lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

void zlatm2 (lapack_complex_double *res, lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);

For gfortran (libmkl_gf_*) interface libraries:
lapack_complex_float clatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

lapack_complex_double zlatm2 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the Intel® MKL Developer Guide.

Include Files

- mkl.h

Description

The ?latm2 routine returns entry \((i, j)\) of a random matrix of dimension \((m, n)\). It is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm2 differs from ?latm3 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm2, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3 can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers).

The matrix whose \((i, j)\) entry is returned is constructed as follows (this routine only computes one entry):

- If \(i\) is outside \((1..m)\) or \(j\) is outside \((1..n)\), returns zero (this is convenient for generating matrices in band format).
- Generate a matrix \(A\) with random entries of distribution \(idist\).
- Set the diagonal to \(D\).
- Grade the matrix, if desired, from the left (by \(dl\)) and/or from the right (by \(dr\) or \(dl\)) as specified by \(igrade\).
- Permute, if desired, the rows and/or columns as specified by \(ipvtng\) and \(iwork\).
- Band the matrix to have lower bandwidth \(kl\) and upper bandwidth \(ku\).
- Set random entries to zero as specified by \(sparse\).

Input Parameters

\(m\)  
Number of rows of the matrix.

\(n\)  
Number of columns of the matrix.

\(i\)  
Row of the entry to be returned.

\(j\)  
Column of the entry to be returned.

\(kl\)  
Lower bandwidth.

\(ku\)  
Upper bandwidth.

\(idist\)  
On entry, \(idist\) specifies the type of distribution to be used to generate a random matrix.

for slatm2 and dlatm2:
= 1: uniform (0,1)
= 2: uniform (-1,1)
= 3: normal (0,1)

for clatm2 and zlatm2:
= 1: real and imaginary parts each uniform (0,1)
= 2: real and imaginary parts each uniform (-1,1)
= 3: real and imaginary parts each normal (0,1)
= 4: complex number uniform in disk (0, 1)

**iseed**
Array, size 4.
Seed for the random number generator.

**d**
Array, size \( \min(i, j) \). Diagonal entries of matrix.

**igrade**
Specifies grading of matrix as follows:
= 0: no grading
= 1: matrix premultiplied by \( \text{diag}(dl) \)
= 2: matrix postmultiplied by \( \text{diag}(dr) \)
= 3: matrix premultiplied by \( \text{diag}(dl) \) and postmultiplied by \( \text{diag}(dr) \)
= 4: matrix premultiplied by \( \text{diag}(dl) \) and postmultiplied by \( \text{inv}(\text{diag}(dl)) \)

For slatm2 and slatm2:
= 5: matrix premultiplied by \( \text{diag}(dl) \) and postmultiplied by \( \text{diag}(dl) \)

For clatm2 and zlatm2:
= 5: matrix premultiplied by \( \text{diag}(dl) \) and postmultiplied by \( \text{diag(\text{conjg}(dl))} \)
= 6: matrix premultiplied by \( \text{diag}(dl) \) and postmultiplied by \( \text{diag}(dl) \)

**dl**
Array, size \((i \text{ or } j)\), as appropriate.
Left scale factors for grading matrix.

**dr**
Array, size \((i \text{ or } j)\), as appropriate.
Right scale factors for grading matrix.

**ipvtng**
On entry specifies pivoting permutations as follows:
= 0: none
= 1: row pivoting
= 2: column pivoting
= 3: full pivoting, i.e., on both sides

**iwork**
Array, size \((i \text{ or } j)\), as appropriate. This array specifies the permutation used. The row (or column) in position \(k\) was originally in position \(iwork[k-1]\). This differs from \(\text{iwork}\) for ?latm3.
Specifies the sparsity of the matrix. If sparse matrix is to be generated, sparse should lie between 0 and 1. A uniform \((0, 1)\) random number \(x\) is generated and compared to \(\text{sparse}\). If \(x\) is larger the matrix entry is unchanged and if \(x\) is smaller the entry is set to zero. Thus on the average a fraction \(\text{sparse}\) of the entries will be set to zero.

**Output Parameters**

\(\text{iseed}\)

On exit, the seed is updated.

**Return Values**

The function returns an entry of a random matrix (for complex variations `libmkl_gf_*` interface layer/libraries return the result as the parameter \(\text{res}\)).

**slatm3**

Returns set entry of a random matrix.

**Syntax**

```c
float slatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, float *d, lapack_int *igrade, float *dl, float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

double dlatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, double *d, lapack_int *igrade, double *dl, double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);
```

The data types for complex variations depend on whether or not the application links with Gnu Fortran (gfortran) libraries.

**For non-gfortran (libmkl_intel_*) interface libraries:**

```c
void clatm3 (lapack_complex_float *res, lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);

void zlatm3 (lapack_complex_double *res, lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);
```

**For gfortran (libmkl_gf_*) interface libraries:**

```c
lapack_complex_float clatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_float *d, lapack_int *igrade, lapack_complex_float *dl, lapack_complex_float *dr, lapack_int *ipvtng, lapack_int *iwork, float *sparse);
```
lapack_complex_double zlatm3 (lapack_int *m, lapack_int *n, lapack_int *i, lapack_int *j, lapack_int *isub, lapack_int *jsub, lapack_int *kl, lapack_int *ku, lapack_int *idist, lapack_int *iseed, lapack_complex_double *d, lapack_int *igrade, lapack_complex_double *dl, lapack_complex_double *dr, lapack_int *ipvtng, lapack_int *iwork, double *sparse);

To understand the difference between the non-gfortran and gfortran interfaces and when to use each of them, see Dynamic Libraries in the lib/intel64 Directory in the Intel® MKL Developer Guide.

Include Files
• mkl.h

Description
The ?latm3 routine returns the (isub, jsub) entry of a random matrix of dimension (m, n) described by the other parameters. (isub, jsub) is the final position of the (i, j) entry after pivoting according to ipvtng and iwork. ?latm3 is called by the ?latmr routine in order to build random test matrices. No error checking on parameters is done, because this routine is called in a tight loop by ?latmr which has already checked the parameters.

Use of ?latm3 differs from ?latm2 in the order in which the random number generator is called to fill in random matrix entries. With ?latm2, the generator is called to fill in the pivoted matrix columnwise. With ?latm3, the generator is called to fill in the matrix columnwise, after which it is pivoted. Thus, ?latm3 can be used to construct random matrices which differ only in their order of rows and/or columns. ?latm2 is used to construct band matrices while avoiding calling the random number generator for entries outside the band (and therefore generating random numbers in different orders for different pivot orders).

The matrix whose (isub, jsub) entry is returned is constructed as follows (this routine only computes one entry):
• If isub is outside (1..m) or jsub is outside (1..n), returns zero (this is convenient for generating matrices in band format).
• Generate a matrix A with random entries of distribution idist.
• Set the diagonal to D.
• Grade the matrix, if desired, from the left (by dl) and/or from the right (by dr or dl) as specified by igrade.
• Permute, if desired, the rows and/or columns as specified by ipvtng and iwork.
• Band the matrix to have lower bandwidth kl and upper bandwidth ku.
• Set random entries to zero as specified by sparse.

Input Parameters

m
Number of rows of matrix.

n
Number of columns of matrix.

i
Row of unpivoted entry to be returned.

j
Column of unpivoted entry to be returned.

isub
Row of pivoted entry to be returned.

jsub
Column of pivoted entry to be returned.

kl
Lower bandwidth.

ku
Upper bandwidth.
**idist**

On entry, idist specifies the type of distribution to be used to generate a random matrix.

For slatm2 and dlatm2:
- $1$: uniform $(0,1)$
- $2$: uniform $(-1,1)$
- $3$: normal $(0,1)$

For clatm2 and zlatm2:
- $1$: real and imaginary parts each uniform $(0,1)$
- $2$: real and imaginary parts each uniform $(-1,1)$
- $3$: real and imaginary parts each normal $(0,1)$
- $4$: complex number uniform in disk$(0, 1)$

**iseed**

Array, size 4.

Seed for random number generator.

**d**

Array, size $(\min(i, j))$. Diagonal entries of matrix.

**igrade**

Specifies grading of matrix as follows:
- $0$: no grading
- $1$: matrix premultiplied by $\text{diag}(dl)$
- $2$: matrix postmultiplied by $\text{diag}(dr)$
- $3$: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dr)$
- $4$: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{inv}(\text{diag}(dl))$

For slatm2 and slatm2:
- $5$: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(di)$

For clatm2 and zlatm2:
- $5$: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(\text{conjg}(dl))$
- $6$: matrix premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$

**dl**

Array, size $(i \text{ or } j$, as appropriate). Left scale factors for grading matrix.

**dr**

Array, size $(i \text{ or } j$, as appropriate). Right scale factors for grading matrix.

**ipvtng**

On entry specifies pivoting permutations as follows:
- If $ipvtng = 0$: none.
- If $ipvtng = 1$: row pivoting.
- If $ipvtng = 2$: column pivoting.
- If $ipvtng = 3$: full pivoting, i.e., on both sides.
On entry, specifies the sparsity of the matrix if sparse matrix is to be generated. *sparse* should lie between 0 and 1. A uniform(0, 1) random number *x* is generated and compared to *sparse*; if *x* is larger the matrix entry is unchanged and if *x* is smaller the entry is set to zero. Thus on the average a fraction *sparse* of the entries will be set to zero.

Array, size (i or j, as appropriate). This array specifies the permutation used. The row (or column) originally in position *k* is in position *iwork*[k - 1] after pivoting. This differs from iwork for ?latm2.

**Output Parameters**

**isub**

On exit, row of pivoted entry is updated.

**jsub**

On exit, column of pivoted entry is updated.

**iseed**

On exit, the seed is updated.

**Return Values**

The function returns an entry of a random matrix (for complex variations libmkl_gf_* interface layer/libraries return the result as the parameter *res*).

**?latm5**

Generates matrices involved in the Generalized Sylvester equation.

**Syntax**

```c
void slatm5 (*prtype, lapack_int *m, lapack_int *n, float *a, lapack_int *lda, float *b, lapack_int *ldb, float *c, lapack_int *ldc, float *d, lapack_int *ldd, float *e, lapack_int *lde, float *f, lapack_int *ldf, float *r, lapack_int *ldr, float *l, lapack_int *ldl, float *alpha, lapack_int *qblcka, lapack_int *qblckb);
void dlatm5 (*prtype, lapack_int *m, lapack_int *n, double *a, lapack_int *lda, double *b, lapack_int *ldb, double *c, lapack_int *ldc, double *d, lapack_int *ldd, double *e, lapack_int *lde, double *f, lapack_int *ldf, double *r, lapack_int *ldr, double *l, lapack_int *ldl, double *alpha, lapack_int *qblcka, lapack_int *qblckb);
void clatm5 (*prtype, lapack_int *m, lapack_int *n, lapack_complex_float *a, lapack_int *lda, lapack_complex_float *b, lapack_int *ldb, lapack_complex_float *c, lapack_int *ldc, lapack_complex_float *d, lapack_int *ldd, lapack_complex_float *e, lapack_int *lde, lapack_complex_float *f, lapack_int *ldf, lapack_complex_float *r, lapack_int *ldr, lapack_complex_float *l, lapack_int *ldl, float *alpha, lapack_int *qblcka, lapack_int *qblckb);
void zlatm5 (*prtype, lapack_int *m, lapack_int *n, lapack_complex_double *a, lapack_int *lda, lapack_complex_double *b, lapack_int *ldb, lapack_complex_double *c, lapack_int *ldc, lapack_complex_double *d, lapack_int *ldd, lapack_complex_double *e, lapack_int *lde, lapack_complex_double *f, lapack_int *ldf, lapack_complex_double *r, lapack_int *ldr, lapack_complex_double *l, lapack_int *ldl, float *alpha, lapack_int *qblcka, lapack_int *qblckb);
```

**Include Files**

- mkl.h
Description

The \texttt{latm5} routine generates matrices involved in the Generalized Sylvester equation:

\[
A \ast R - L \ast B = C \\
D \ast R - L \ast E = F
\]

They also satisfy the diagonalization condition:

\[
\begin{bmatrix}
I & -L \\
I & B
\end{bmatrix}
\begin{bmatrix}
A & R \\
I & I
\end{bmatrix} = 
\begin{bmatrix}
A \\
B
\end{bmatrix}
\]

\[
\begin{bmatrix}
I & -L \\
I & E
\end{bmatrix}
\begin{bmatrix}
D & R \\
I & I
\end{bmatrix} = 
\begin{bmatrix}
D \\
E
\end{bmatrix}
\]

Input Parameters

\texttt{prtype}

- Specifies the type of matrices to generate.
  - If \texttt{prtype} = 1, \( A \) and \( B \) are Jordan blocks, \( D \) and \( E \) are identity matrices.
    
    \( A \):
    
    If \((i == j)\) then \( A_{i,j} = 1.0 \).
    
    If \((j == i + 1)\) then \( A_{i,j} = -1.0 \).
    
    Otherwise \( A_{i,j} = 0.0, i, j = 1...m \)
    
    \( B \):
    
    If \((i == j)\) then \( B_{i,j} = 1.0 - \alpha \).
    
    If \((j == i + 1)\) then \( B_{i,j} = 1.0 \).
    
    Otherwise \( B_{i,j} = 0.0, i, j = 1...n \).
    
    \( D \):
    
    If \((i == j)\) then \( D_{i,j} = 1.0 \).
    
    Otherwise \( D_{i,j} = 0.0, i, j = 1...m \).
    
    \( E \):
    
    If \((i == j)\) then \( E_{i,j} = 1.0 \).
    
    Otherwise \( E_{i,j} = 0.0, i, j = 1...n \).
  - \( L = R \) are chosen from \([-10...10]\), which specifies the right hand sides \((C, F)\).
  - If \texttt{prtype} = 2 or 3: Triangular and/or quasi- triangular.
    
    \( A \):
    
    If \((i \leq j)\) then \( A_{i,j} = [-1...1] \).
    
    Otherwise \( A_{i,j} = 0.0, i, j = 1...M \).
    
    If \( \texttt{prtype} = 3 \) then \( A_{k+1,k+1} = A_{k,k} \);
    
    \( A_{k+1,k} = [-1...1] \); 
    
    \( \text{sign}(A_{k,k+1}) = -(\text{sign}(A_{k+1,k})) \).
\( k = 1, \ m-1, \ qblcka \)

\( B : \)
If \((i \leq j)\) then \(B_{i,j} = [-1...1]\).
Otherwise \(B_{i,j} = 0.0, \ i, j = 1...n.\)
If \(\text{prtype} = 3\) then \(B_{k+1,k+1} = B_{k,k}\)
\(B_{k+1,k} = [-1...1]\)
\(\text{sign}(B_{k,k+1}) = -(\text{sign}(B_{k+1,k}))\)
\(k = 1, n - 1, \ qblckb.\)

\( D : \)
If \((i \leq j)\) then \(D_{i,j} = [-1...1]\).
Otherwise \(D_{i,j} = 0.0, \ i, j = 1...m.\)

\( E : \)
If \((i \leq j)\) then \(E_{i,j} = [-1...1]\).
Otherwise \(E_{i,j} = 0.0, \ i, j = 1...n.\)

L, R are chosen from \([-10...10]\), which specifies the right hand sides \((C, F)\).

- If \(\text{prtype} = 4\) Full
  \(A_{i,j} = [-10...10]\)
  \(D_{i,j} = [-1...1] \ i, j = 1...m\)
  \(B_{i,j} = [-10...10]\)
  \(E_{i,j} = [-1...1] \ i, j = 1...n\)
  \(R_{i,j} = [-10...10]\)
  \(L_{i,j} = [-1...1] \ i = 1..m \ j = 1...n\)

\( L \) and \( R \) specifies the right hand sides \((C, F)\).

- If \(\text{prtype} = 5\) special case common and/or close eigs.

\( m \)
Specifies the order of \(A\) and \(D\) and the number of rows in \(C, F, R\) and \(L\).

\( n \)
Specifies the order of \(B\) and \(E\) and the number of columns in \(C, F, R\) and \(L\).

\( lda \)
The leading dimension of \(a\).

\( ldb \)
The leading dimension of \(b\).

\( ldc \)
The leading dimension of \(c\).

\( ldd \)
The leading dimension of \(d\).

\( lde \)
The leading dimension of \(e\).

\( ldf \)
The leading dimension of \(f\).

\( ldr \)
The leading dimension of \(r\).

\( ldl \)
The leading dimension of \(l\).
**SLATM6**
Generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

### Syntax

```c
void slatm6 (lapack_int *type, lapack_int *n, float *a, lapack_int *lda, float *b, float *x, lapack_int *ldx, float *y, lapack_int *ldy, float *alpha, float *beta, float *wx, float *wy, float *s, float *dif);
void dlatm6 (lapack_int *type, lapack_int *n, double *a, lapack_int *lda, double *b, double *x, lapack_int *ldx, double *y, lapack_int *ldy, double *alpha, double *beta, double *wx, double *wy, double *s, double *dif);
void clatm6 (lapack_int *type, lapack_int *n, lapack_complex_float *a, lapack_int *lda, lapack_complex_float *b, lapack_complex_float *x, lapack_int *ldx, lapack_complex_float *y, lapack_int *ldy, lapack_complex_float *alpha, lapack_complex_float *beta, lapack_complex_float *wx, lapack_complex_float *wy, float *s, float *dif);
```
void zlatm6 (lapack_int *type, lapack_int *n, lapack_complex_double *a, lapack_int *lda, lapack_complex_double *b, lapack_complex_double *x, lapack_int *ldx, lapack_complex_double *y, lapack_int *ldy, lapack_complex_double *alpha, lapack_complex_double *beta, lapack_complex_double *wx, lapack_complex_double *wy, double *s, double *dif);

Include Files

- mkl.h

Description

The zlatm6 routine generates test matrices for the generalized eigenvalue problem, their corresponding right and left eigenvector matrices, and also reciprocal condition numbers for all eigenvalues and the reciprocal condition numbers of eigenvectors corresponding to the 1th and 5th eigenvalues.

There two kinds of test matrix pairs:

\[(A, B) = \text{inverse}(YH) \times (Da, Db) \times \text{inverse}(X)\]

Type 1:

\[
Da = \begin{bmatrix}
1+a & 0 & 0 & 0 & 0 \\
0 & 2+a & 0 & 0 & 0 \\
0 & 0 & 3+a & 0 & 0 \\
0 & 0 & 0 & 4+a & 0 \\
0 & 0 & 0 & 0 & 5+2
\end{bmatrix}, \quad Db = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Type 2:

\[
Da = \begin{bmatrix}
1+i & 0 & 0 & 0 & 0 \\
0 & 1-i & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & (1+a) + (1+b)i & 0 \\
0 & 0 & 0 & 0 & (1+a) - (1+b)i
\end{bmatrix}, \quad Db = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

In both cases the same inverse(YH) and inverse(X) are used to compute (A, B), giving the exact eigenvectors to (A,B) as (YH, X):

\[
YH = \begin{bmatrix}
1 & 0 & -y & y & -y \\
0 & 1 & -y & y & -y \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}, \quad X = \begin{bmatrix}
1 & 0 & -x & -x & x \\
0 & 1 & x & -x & -x \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1
\end{bmatrix},
\]

where a, b, x and y will have all values independently of each other.

Input Parameters

- **type**
  - Specifies the problem type.

- **n**
  - Size of the matrices A and B.
lda
The leading dimension of $a$ and of $b$.

ldx
The leading dimension of $x$.

ldy
The leading dimension of $y$.

alpha, beta
Weighting constants for matrix $A$.

wx
Constant for right eigenvector matrix.

wy
Constant for left eigenvector matrix.

Output Parameters

$a$
Array, size $lda*n$. On exit, $a$ contains the $n$-by-$n$ matrix initialized according to type.

$b$
Array, size $lda*n$. On exit, $b$ contains the $n$-by-$n$ matrix initialized according to type.

$x$
Array, size $ldx*n$. On exit, $x$ contains the $n$-by-$n$ matrix of right eigenvectors.

$y$
Array, size $ldy*n$. On exit, $y$ is the $n$-by-$n$ matrix of left eigenvectors.

$s$
Array, size $(n)$. $s[i-1]$ is the reciprocal condition number for eigenvalue $i$.

$\text{dif}$
Array, size$(n)$. $\text{dif}[i-1]$ is the reciprocal condition number for eigenvector $i$.

\texttt{?latme}
\textit{Generates random non-symmetric square matrices with specified eigenvalues.}

\textbf{Syntax}

void slatme (lapack_int *n, char *dist, lapack_int *iseed, float *d, lapack_int *mode, float *cond, float *dmax, char *ei, char *rsign, char *upper, char *sim, float *ds, lapack_int *modes, float *conds, lapack_int *kl, lapack_int *ku, float *anorm, float *a, lapack_int *lda, float *work, lapack_int *info);

double dmax, char *ei, char *rsign, char *upper, char *sim, double *ds, lapack_int *modes, double *conds, lapack_int *kl, lapack_int *ku, double *anorm, lapack_int *lda, lapack_int *work, lapack_int *info);
Include Files

- mkl.h

Description

The ?latme routine generates random non-symmetric square matrices with specified eigenvalues. ?latme operates by applying the following sequence of operations:

1. Set the diagonal to \( d \), where \( d \) may be input or computed according to mode, cond, dmax, and rsign as described below.
2. If upper = 'T', the upper triangle of \( a \) is set to random values out of distribution dist.
3. If sim='T', \( a \) is multiplied on the left by a random matrix \( X \), whose singular values are specified by \( ds \), modes, and conds, and on the right by \( X \) inverse.
4. If \( kl < n-1 \), the lower bandwidth is reduced to \( kl \) using Householder transformations. If \( ku < n-1 \), the upper bandwidth is reduced to \( ku \).
5. If anorm is not negative, the matrix is scaled to have maximum-element-norm anorm.

NOTE
Since the matrix cannot be reduced beyond Hessenberg form, no packing options are available.

Input Parameters

\( n \)

The number of columns (or rows) of \( A \).

\( dist \)

On entry, \( dist \) specifies the type of distribution to be used to generate the random eigen-/singular values, and on the upper triangle (see upper).

- If \( dist = 'U' \): uniform( 0, 1 )
- If \( dist = 'S' \): uniform( -1, 1 )
- If \( dist = 'N' \): normal( 0, 1 )
- If \( dist = 'D' \): uniform on the complex disc \( |z| < 1 \).

\( iseed \)

Array, size 4.

On entry \( iseed \) specifies the seed of the random number generator. The elements should lie between 0 and 4095 inclusive, and \( iseed[3] \) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.

\( d \)

Array, size \((n)\). This array is used to specify the eigenvalues of \( A \).

If mode = 0, then \( d \) is assumed to contain the eigenvalues. Otherwise they are computed according to mode, cond, dmax, and rsign and placed in \( d \).

\( mode \)

On entry \( mode \) describes how the eigenvalues are to be specified:

- \( mode = 0 \) means use \( d \) (with \( ei \) for slatme and dlatme) as input.
- \( mode = 1 \) sets \( d[0] = 1 \) and \( d[2:n]=1.0/cond \).
- \( mode = 2 \) sets \( d[0:n-2] = 1 \) and \( d[n-1]=1.0/cond \).
- \( mode = 3 \) sets \( d[i - 1] = cond**((-i-1)/(n-1)) \).
mode = 4 sets $d[i - 1] = 1 - (i-1)/(n-1)*(1 - 1/\text{cond})$.

mode = 5 sets $d$ to random numbers in the range $(1/\text{cond}, 1)$ such that their logarithms are uniformly distributed.

mode = 6 sets $d$ to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as $\text{abs}(\text{mode})$, except that the order of the elements of $d$ is reversed.

Thus if mode is between 1 and 4, $d$ has entries ranging from 1 to $1/\text{cond}$, if between -1 and -4, $d$ has entries ranging from $1/\text{cond}$ to 1.

On entry, this is used as described under mode above. If used, it must be $\geq 1$.

If mode is not -6, 0 or 6, the contents of $d$ as computed according to mode and cond are scaled by $dmax / \max(\text{abs}(d[i - 1]))$. Note that dmax needs not be positive or real: if dmax is negative or complex (or zero), $d$ will be scaled by a negative or complex number (or zero). If $rsign = 'F'$ then the largest (absolute) eigenvalue will be equal to $dmax$.

$cond$ used by slatme and dlatme only.

Array, size $(n)$.

If mode = 0, and $ei[0]$ is not ' ' (space character), this array specifies which elements of $d$ (on input) are real eigenvalues and which are the real and imaginary parts of a complex conjugate pair of eigenvalues. The elements of $ei$ may then only have the values 'R' and 'I'.

If $ei[j - 1] = 'R'$ and $ei[j] = 'I'$, then the $j$-th eigenvalue is $\text{cmplx}(d[j - 1], d[j])$, and the $(j+1)$-th is the complex conjugate thereof.

If $ei[j - 1] = ei[j] = 'R'$, then the $j$-th eigenvalue is $d[j - 1]$ (i.e., real). $ei[0]$ may not be 'I', nor may two adjacent elements of $ei$ both have the value 'I'.

If mode is not 0, then $ei$ is ignored. If mode is 0 and $ei[0] = ' ' $, then the eigenvalues will all be real.

$rsign$ if mode is not 0, 6, or -6, and $rsign = 'T'$, then the elements of $d$, as computed according to mode and cond, are multiplied by a random sign (+1 or -1) for slatme and dlatme or by a complex number from the unit circle $|z| = 1$ for clatme and zlatme.

If $rsign = 'F'$, the elements of $d$ are not multiplied. $rsign$ may only have the values 'T' or 'F'.

$upper$ if upper = 'T', then the elements of $a$ above the diagonal will be set to random numbers out of dist.

If upper = 'F', they will not. upper may only have the values 'T' or 'F'.

$sim$ if sim = 'T', then $a$ will be operated on by a "similarity transform", i.e., multiplied on the left by a matrix $X$ and on the right by $X$ inverse. $X = USV$, where $U$ and $V$ are random unitary matrices and $S$ is a (diagonal) matrix of singular values specified by $ds$, modes, and conds.
If sim = 'F', then a will not be transformed.

This array is used to specify the singular values of X, in the same way that
d specifies the eigenvalues of a. If mode = 0, the ds contains the singular
values, which may not be zero.

Similar to mode, but for specifying the diagonal of S. modes = -6 and +6
are not allowed (since they would result in randomly ill-conditioned
eigenvalues.)

Similar to cond, but for specifying the diagonal of S.

This specifies the lower bandwidth of the matrix. kl = 1 specifies upper
Hessenberg form. If kl is at least n-1, then A will have full lower
bandwidth.

This specifies the upper bandwidth of the matrix. ku = 1 specifies lower
Hessenberg form.

If ku is at least n-1, then a will have full upper bandwidth.

If ku and ku are both at least n-1, then a will be dense. Only one of ku and
kl may be less than n-1.

If anorm is not negative, then a is scaled by a non-negative real number to
make the maximum-element-norm of a to be anorm.

Number of rows of matrix A.

Array, size (3*n). Workspace.

On exit, the seed is updated.

Modified if mode is nonzero.

Modified if mode is nonzero.

Array, size lda*n. On exit, a is the desired test matrix.

If info = 0, execution is successful.

If info = -1, n is negative.

If info = -2, dist is an illegal string.

If info = -5, mode is not in range -6 to 6.

If info = -6, cond is less than 1.0, and mode is not -6, 0, or 6.

If info = -9, rsign is not 'T' or 'F'.

If info = -10, upper is not 'T' or 'F'.

If info = -11, sim is not 'T' or 'F'.

If info = -12, modes = 0 and ds has a zero singular value.

If info = -13, modes is not in the range -5 to 5.

If info = -14, modes is nonzero and conds is less than 1.
If info = -15, kl is less than 1.
If info = -16, ku is less than 1, or kl and ku are both less than n-1.
If info = -19, lda is less than m.
If info = 1, error return from ?latml (computing d).
If info = 2, cannot scale to dmax (max. eigenvalue is 0).
If info = 3, error return from slatml (for slatme and clatme), dlatml (for dlatme and zlatme).
If info = 4, error return from ?large.
If info = 5, zero singular value from slatml (for slatme and clatme), dlatml (for dlatme and zlatme).

?latmr

Generates random matrices of various types.

**Syntax**

```c
void slatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym, float *d, lapack_int *mode, float *cond, float *dmax, char *rsign, char *grade, float *dl, lapack_int *model, float *condl, float *dr, lapack_int *moder, float *condr, char *pivtn, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, float *sparse, float *anorm, char *pack, float *a, lapack_int *lda, lapack_int *iwork, lapack_int *info);
void dlatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym, char *dl, lapack_int *mode, double *cond, double *dmax, char *rsign, char *grade, double *dl, lapack_int *model, double *condl, double *dr, lapack_int *moder, double *condr, char *pivtn, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, double *sparse, double *anorm, char *pack, double *a, lapack_int *lda, lapack_int *iwork, lapack_int *info);
void clatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym, lapack_complex _d, lapack_int *mode, float *cond, lapack_complex *dmax, char *rsign, char *grade, lapack_complex _d, lapack_int *model, float *condl, lapack_complex _dr, lapack_int *moder, float *condr, char *pivtn, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, float *sparse, float *anorm, char *pack, float *a, lapack_int *lda, lapack_int *iwork, lapack_int *info);
void zlatmr (lapack_int *m, lapack_int *n, char *dist, lapack_int *iseed, char *sym, lapack_complex_double _d, lapack_int *mode, float *cond, lapack_complex_double *dmax, char *rsign, char *grade, lapack_complex_double _d, lapack_int *model, float *condl, lapack_complex_double _dr, lapack_int *moder, float *condr, char *pivtn, lapack_int *ipivot, lapack_int *kl, lapack_int *ku, float *sparse, float *anorm, char *pack, float *a, lapack_int *lda, lapack_int *iwork, lapack_int *info);
```

**Description**

The ?latmr routine operates by applying the following sequence of operations:

1. Generate a matrix $A$ with random entries of distribution $\text{dist}$:
   - If $\text{sym} = 'S'$, the matrix is symmetric,
   - If $\text{sym} = 'H$', the matrix is Hermitian,
If \( \text{sym} = 'N' \), the matrix is nonsymmetric.

2. Set the diagonal to \( D \), where \( D \) may be input or computed according to \( \text{mode} \), \( \text{cond} \), \( \text{dmax} \) and \( \text{rsign} \) as described below.

3. Grade the matrix, if desired, from the left or right as specified by grade. The inputs \( \text{dl} \), \( \text{modl} \), \( \text{condl} \), \( \text{dr} \), \( \text{modr} \) and \( \text{condr} \) also determine the grading as described below.

4. Permute, if desired, the rows and/or columns as specified by \( \text{pivtng} \) and \( \text{ipivot} \).

5. Make \( A \) a band matrix, if desired, by zeroing out the matrix outside a band of lower bandwidth \( kl \) and upper bandwidth \( ku \).

6. Scale \( A \), if desired, to have maximum entry \( \text{anorm} \).

7. Pack the matrix if desired. See options specified by the \( \text{pack} \) parameter.

---

**NOTE**

If two calls to \(?latmr\) differ only in the \( \text{pack} \) parameter, they generate mathematically equivalent matrices. If two calls to \(?latmr\) both have full bandwidth \((kl = m-1 \text{ and } ku = n-1)\), and differ only in the \( \text{pivtng} \) and \( \text{pack} \) parameters, then the matrices generated differ only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be and is not maintained with less than full bandwidth.

---

**Input Parameters**

\( m \)  
Number of rows of \( A \).

\( n \)  
Number of columns of \( A \).

\( \text{dist} \)  
On entry, \( \text{dist} \) specifies the type of distribution to be used to generate a random matrix.

- If \( \text{dist} = 'U' \), real and imaginary parts are independent uniform\(( 0, 1 )\).
- If \( \text{dist} = 'S' \), real and imaginary parts are independent uniform\(( -1, 1 )\).
- If \( \text{dist} = 'N' \), real and imaginary parts are independent normal\(( 0, 1 )\).
- If \( \text{dist} = 'D' \), distribution is uniform on interior of unit disk.

\( \text{iseed} \)  
Array, size 4.

On entry, \( \text{iseed} \) specifies the seed of the random number generator. They should lie between 0 and 4095 inclusive, and \( \text{iseed}[3] \) should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers.

\( \text{sym} \)  
If \( \text{sym} = 'S' \), generated matrix is symmetric.

If \( \text{sym} = 'H' \), generated matrix is Hermitian.

If \( \text{sym} = 'N' \), generated matrix is nonsymmetric.

\( d \)  
On entry this array specifies the diagonal entries of the diagonal of \( A \). \( d \) may either be specified on entry, or set according to \( \text{mode} \) and \( \text{cond} \) as described below. If the matrix is Hermitian, the real part of \( d \) is taken. May be changed on exit if \( \text{mode} \) is nonzero.

\( \text{mode} \)  
On entry describes how \( d \) is to be used:
mode = 0 means use $d$ as input.

mode = 1 sets $d[0]=1$ and $d[1:n-1]=1.0/\text{cond}$.

mode = 2 sets $d[0:n-2]=1$ and $d[n-1]=1.0/\text{cond}$.

mode = 3 sets $d[1-i]=\text{cond}^*\left(-1-i/(n-1)\right)$.

mode = 4 sets $d[1-i]=1 - (1-i)/(n-1)*(1 - 1/\text{cond})$.

mode = 5 sets $d$ to random numbers in the range $(1/\text{cond}, 1)$ such that their logarithms are uniformly distributed.

mode = 6 sets $d$ to random numbers from same distribution as the rest of the matrix.

mode < 0 has the same meaning as $\text{abs}(\text{mode})$, except that the order of the elements of $d$ is reversed.

Thus if mode is between 1 and 4, $d$ has entries ranging from 1 to $1/\text{cond}$, if between -1 and -4, $D$ has entries ranging from $1/\text{cond}$ to 1.

On entry, used as described under mode above. If used, $\text{cond}$ must be $\geq 1$.

If mode is not -6, 0, or 6, the diagonal is scaled by $d_{\text{max}} / \max(\text{abs}(d[i]))$, so that maximum absolute entry of diagonal is $\text{abs}(d_{\text{max}})$. If $d_{\text{max}}$ is complex (or zero), the diagonal is scaled by a complex number (or zero).

If mode is not -6, 0, or 6, specifies the sign of the diagonal as follows:

For slatmr and dlatmr, if $rsign = 'T'$, diagonal entries are multiplied 1 or -1 with a probability of 0.5.

For clatmr and zlatmr, if $rsign = 'T'$, diagonal entries are multiplied by a random complex number uniformly distributed with absolute value 1.

If $rsign = 'F'$, diagonal entries are unchanged.

Specifies grading of matrix as follows:

If grade = 'N', there is no grading

If grade = 'L', matrix is premultiplied by $\text{diag}(dl)$ (only if matrix is nonsymmetric)

If grade = 'R', matrix is postmultiplied by $\text{diag}(dr)$ (only if matrix is nonsymmetric)

If grade = 'B', matrix is premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dr)$ (only if matrix is nonsymmetric)

If grade = 'H', matrix is premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(\text{conjg}(dl))$ (only if matrix is Hermitian or nonsymmetric)

If grade = 'S', matrix is premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{diag}(dl)$ (only if matrix is symmetric or nonsymmetric)

If grade = 'E', matrix is premultiplied by $\text{diag}(dl)$ and postmultiplied by $\text{inv(\text{diag}(dl))}$ (only if matrix is nonsymmetric)
NOTE
if grade = 'E', then m must equal n.

dl
Array, size (m).
If model = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under grade above.
If model is not zero, then dl is set according to model and condl, analogous to the way D is set according to mode and cond (except there is no dmax parameter for dl).
If grade = 'E', then dl cannot have zero entries.
Not referenced if grade = 'N' or 'R'. Changed on exit.

model
This specifies how the diagonal array dl is computed, just as mode specifies how D is computed.

condl
When model is not zero, this specifies the condition number of the computed dl.

dr
If moder = 0, then on entry this array specifies the diagonal entries of a diagonal matrix used as described under grade above.
If moder is not zero, then dr is set according to moder and condr, analogous to the way d is set according to mode and cond (except there is no dmax parameter for dr).
Not referenced if grade = 'N', 'L', 'H', 'S' or 'E'.

moder
This specifies how the diagonal array dr is to be computed, just as mode specifies how d is to be computed.

condr
When moder is not zero, this specifies the condition number of the computed dr.

pivtng
On entry specifies pivoting permutations as follows:
If pivtng = 'N' or ' ': no pivoting permutation.
If pivtng = 'L': left or row pivoting (matrix must be nonsymmetric).
If pivtng = 'R': right or column pivoting (matrix must be nonsymmetric).
If pivtng = 'B' or 'F': both or full pivoting, i.e., on both sides. In this case, m must equal n.
If two calls to ?latmr both have full bandwidth (ki = m - 1 and ku = n-1), and differ only in the pivtng and pack parameters, then the matrices generated differs only in the order of the rows and columns, and otherwise contain the same data. This consistency cannot be maintained with less than full bandwidth.

ipivot
Array, size (n or m) This array specifies the permutation used. After the basic matrix is generated, the rows, columns, or both are permuted.
If row pivoting is selected, ?latmr starts with the last row and interchanges row \( m \) and row \( \text{ipivot}[m - 1] \), then moves to the next-to-last row, interchanging rows \( [m - 2] \) and row \( \text{ipivot}[m - 2] \), and so on. In terms of "2-cycles", the permutation is \((1 \text{ipivot}[0]) (2 \text{ipivot}[1]) \ldots (\text{mipivot}[m - 1])\) where the rightmost cycle is applied first. This is the inverse of the effect of pivoting in LINPACK. The idea is that factoring (with pivoting) an identity matrix which has been inverse-pivoted in this way should result in a pivot vector identical to \( \text{ipivot} \). Not referenced if \( \text{pivtn} = 'N' \).

\( \text{sparse} \)

On entry, specifies the sparsity of the matrix if a sparse matrix is to be generated. \( \text{sparse} \) should lie between 0 and 1. To generate a sparse matrix, for each matrix entry a uniform \((0, 1)\) random number \( x \) is generated and compared to \( \text{sparse} \); if \( x \) is larger the matrix entry is unchanged and if \( x \) is smaller the entry is set to zero. Thus on the average a fraction \( \text{sparse} \) of the entries is set to zero.

\( \text{kl} \)

On entry, specifies the lower bandwidth of the matrix. For example, \( \text{kl} = 0 \) implies upper triangular, \( \text{kl} = 1 \) implies upper Hessenberg, and \( \text{kl} \) at least \( m-1 \) implies the matrix is not banded. Must equal \( \text{ku} \) if matrix is symmetric or Hermitian.

\( \text{ku} \)

On entry, specifies the upper bandwidth of the matrix. For example, \( \text{ku} = 0 \) implies lower triangular, \( \text{ku} = 1 \) implies lower Hessenberg, and \( \text{ku} \) at least \( n-1 \) implies the matrix is not banded. Must equal \( \text{kl} \) if matrix is symmetric or Hermitian.

\( \text{anorm} \)

On entry, specifies maximum entry of output matrix (output matrix is multiplied by a constant so that its largest absolute entry equal \( \text{anorm} \)) if \( \text{anorm} \) is nonnegative. If \( \text{anorm} \) is negative no scaling is done.

\( \text{pack} \)

On entry, specifies packing of matrix as follows:

- If \( \text{pack} = 'N' \): no packing
- If \( \text{pack} = 'U' \): zero out all subdiagonal entries (if symmetric or Hermitian)
- If \( \text{pack} = 'L' \): zero out all superdiagonal entries (if symmetric or Hermitian)
- If \( \text{pack} = 'C' \): store the upper triangle columnwise (only if matrix symmetric or Hermitian or square upper triangular)
- If \( \text{pack} = 'R' \): store the lower triangle columnwise (only if matrix symmetric or Hermitian or square lower triangular) (same as upper half rowwise if symmetric) (same as conjugate upper half rowwise if Hermitian)
- If \( \text{pack} = 'B' \): store the lower triangle in band storage scheme (only if matrix symmetric or Hermitian)
- If \( \text{pack} = 'Q' \): store the upper triangle in band storage scheme (only if matrix symmetric or Hermitian)
- If \( \text{pack} = 'Z' \): store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store \( A \) in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:
### LAPACK storage scheme

<table>
<thead>
<tr>
<th>Value of pack</th>
</tr>
</thead>
<tbody>
<tr>
<td>'Z'</td>
</tr>
<tr>
<td>'B' or 'Q'</td>
</tr>
<tr>
<td>'C' or 'R'</td>
</tr>
</tbody>
</table>

If two calls to `?latmr` differ only in the pack parameter, they generate mathematically equivalent matrices.

#### lda

On entry, `lda` specifies the first dimension of `a` as declared in the calling program.

If `pack` = 'N', 'U' or 'L', `lda` must be at least `max( 1, m )`.

If `pack` = 'C' or 'R', `lda` must be at least 1.

If `pack` = 'B', or 'Q', `lda` must be `min( ku + 1, n )`.

If `pack` = 'Z', `lda` must be at least `kuu + kll + 1`, where `kuu = min( ku, n-1 )` and `kll = min( kl, n-1 )`.

#### iwork

Array, size `(n or m)`. Workspace. Not referenced if `pivtng` = 'N'. Changed on exit.

### Output Parameters

#### iseed

On exit, the seed is changed.

May be changed on exit if `mode` is nonzero.

#### d

On exit, array is changed.

On exit, array is changed.

On exit, `a` is the desired test matrix. Only those entries of `a` which are significant on output is referenced (even if `a` is in packed or band storage format). The unoccupied corners of `a` in band format are zeroed out.

#### info

If `info` = 0, the execution is successful.

If `info` = -1, `m` is negative or unequal to `n` and `sym` = 'S' or 'H'.

If `info` = -2, `n` is negative.

If `info` = -3, `dist` is an illegal string.

If `info` = -5, `sym` is an illegal string.

If `info` = -7, `mode` is not in range -6 to 6.

If `info` = -8, `cond` is less than 1.0, and `mode` is neither -6, 0 nor 6.

If `info` = -10, `mode` is neither -6, 0 nor 6 and `rsign` is an illegal string.

If `info` = -11, `grade` is an illegal string, or `grade` = 'E' and `m` is not equal to `n`, or `grade='L', 'R', 'B', 'S' or 'E' and sym = 'H', or `grade = 'L', 'R', 'B', 'H' or 'E' and sym = 'S'`

If `info` = -12, `grade` = 'E' and `dl` contains zero.
If $\text{info} = -13$, model is not in range -6 to 6 and grade = 'L', 'B', 'H', 'S' or 'E'.

If $\text{info} = -14$, condl is less than 1.0, grade = 'L', 'B', 'H', 'S' or 'E', and model is neither -6, 0 nor 6.

If $\text{info} = -16$, moder is not in range -6 to 6 and grade = 'R' or 'B'.

If $\text{info} = -17$, condr is less than 1.0, grade = 'R' or 'B', and moder is neither -6, 0 nor 6.

If $\text{info} = -18$, pivtng is an illegal string, or pivtng = 'B' or 'F' and $m$ is not equal to $n$, or pivtng = 'L' or 'R' and sym = 'S' or 'H'.

If $\text{info} = -19$, ipivot contains out of range number and pivtng is not equal to 'N'.

If $\text{info} = -20$, kl is negative.

If $\text{info} = -21$, ku is negative, or sym = 'S' or 'H' and ku not equal to kl.

If $\text{info} = -22$, sparse is not in range 0 to 1.

If $\text{info} = -24$, pack is an illegal string, or pack = 'U', 'L', 'B' or 'Q' and sym = 'N', or pack = 'C' and sym = 'N' and either kl is not equal to 0 or n is not equal to $m$, or pack = 'R' and sym = 'N', and either ku is not equal to 0 or n is not equal to $m$.

If $\text{info} = -26$, lda is too small.

If $\text{info} = 1$, error return from ?latm1 (computing $D$).

If $\text{info} = 2$, cannot scale to dmax (max. entry is 0).

If $\text{info} = 3$, error return from ?latm1 (computing dl).

If $\text{info} = 4$, error return from ?latm1 (computing dr).

If $\text{info} = 5$, anorm is positive, but matrix constructed prior to attempting to scale it to have norm anorm, is zero.

?lauum

Computes the product $U^T (U^T)^H$ or $L^H L^T$, where $U$ and $L$ are upper or lower triangular matrices (blocked algorithm).

Syntax

```c
lapack_int LAPACKE_slauum (int matrix_layout , char uplo , lapack_int n , float * a ,
lapack_int lda );

lapack_int LAPACKE_dlauum (int matrix_layout , char uplo , lapack_int n , double * a ,
lapack_int lda );

lapack_int LAPACKE_clauum (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_float * a , lapack_int lda );

lapack_int LAPACKE_zlauum (int matrix_layout , char uplo , lapack_int n ,
lapack_complex_double * a , lapack_int lda );
```
Include Files

- mkl.h

Description

The routine ?lauum computes the product $U^T U$ or $L^T L$ for real flavors, and $U^H U$ or $L^H L$ for complex flavors. Here the triangular factor $U$ or $L$ is stored in the upper or lower triangular part of the array $a$.

If $uplo = 'U'$ or 'u', then the upper triangle of the result is stored, overwriting the factor $U$ in $A$.

If $uplo = 'L'$ or 'l', then the lower triangle of the result is stored, overwriting the factor $L$ in $A$.

This is the blocked form of the algorithm, calling BLAS Level 3 Routines.

Input Parameters

$uplo$

Specifies whether the triangular factor stored in the array $a$ is upper or lower triangular:

- $'U'$: Upper triangular
- $'L'$: Lower triangular

$n$

The order of the triangular factor $U$ or $L$. $n \geq 0$.

$a$

Array of size $\max(1, lda \times n)$.

On entry, the triangular factor $U$ or $L$.

$lda$

The leading dimension of the array $a$. $lda \geq \max(1, n)$.

Output Parameters

$a$

On exit,

- if $uplo = 'U'$, then the upper triangle of $a$ is overwritten with the upper triangle of the product $U^T (U^T U)$;
- if $uplo = 'L'$, then the lower triangle of $a$ is overwritten with the lower triangle of the product $L^T L (L^H L)$.

Return Values

This function returns a value $info$.

- If $info = 0$, the execution is successful.
- If $info = -k$, the $k$-th parameter had an illegal value.
- If $info = -1011$, memory allocation error occurred.

?syswapr

Applies an elementary permutation on the rows and columns of a symmetric matrix.

Syntax

```c
lapack_int LAPACKE_ssyswapr (int matrix_layout, char uplo, lapack_int n, float *a, lapack_int i1, lapack_int i2);
```
lapack_int LAPACKE_dsyswapr (int matrix_layout, char uplo, lapack_int n, double *a, lapack_int i1, lapack_int i2);

lapack_int LAPACKE_csyswapr (int matrix_layout, char uplo, lapack_int n, lapack_complex_float *a, lapack_int i1, lapack_int i2);

lapack_int LAPACKE_zsyswapr (int matrix_layout, char uplo, lapack_int n, lapack_complex_double *a, lapack_int i1, lapack_int i2);

Include Files

- mkl.h

Description

The routine applies an elementary permutation on the rows and columns of a symmetric matrix.

Input Parameters

- **matrix_layout**: Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- **uplo**: Must be 'U' or 'L'. Indicates how the input matrix A has been factored:
  - If uplo = 'U', the array a stores the upper triangular factor U of the factorization $A = U^*D*U$.
  - If uplo = 'L', the array a stores the lower triangular factor L of the factorization $A = L*L^T$.
- **n**: The order of matrix A; $n \geq 0$.
- **nrhs**: The number of right-hand sides; $nrhs \geq 0$.
- **a**: Array of size at least max(1, lda*n).
  - The array a contains the block diagonal matrix D and the multipliers used to obtain the factor U or L as computed by ?sytrf.
- **i1**: Index of the first row to swap.
- **i2**: Index of the second row to swap.

Output Parameters

- **a**
  - If info = 0, the symmetric inverse of the original matrix.
  - If info = 'U', the upper triangular part of the inverse is formed and the part of A below the diagonal is not referenced.
  - If info = 'L', the lower triangular part of the inverse is formed and the part of A above the diagonal is not referenced.

Return Values

This function returns a value info.

- If info = 0, the execution is successful.
- If info = -i, the i-th parameter had an illegal value.
If \( \text{info} = -1011 \), memory allocation error occurred.

**See Also**

?sytrf

**?heswapr**

*Applies an elementary permutation on the rows and columns of a Hermitian matrix.*

**Syntax**

```c
lapack_int LAPACKE_cheswapr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_float* a, lapack_int i1, lapack_int i2);
```

```c
lapack_int LAPACKE_zheswapr (int matrix_layout, char uplo, lapack_int n,
lapack_complex_double* a, lapack_int i1, lapack_int i2);
```

**Include Files**

- mkl.h

**Description**
The routine applies an elementary permutation on the rows and columns of a Hermitian matrix.

**Input Parameters**

- `matrix_layout` specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
- `uplo` must be 'U' or 'L'.
  - Indicates how the input matrix \( A \) has been factored:
    - If `uplo = 'U'`, the array \( a \) stores the upper triangular factor \( U \) of the factorization \( A = U*D*U^H \).
    - If `uplo = 'L'`, the array \( a \) stores the lower triangular factor \( L \) of the factorization \( A = L*D*L^H \).
- `n` the order of matrix \( A \); \( n \geq 0 \).
- `nrhs` the number of right-hand sides; \( nrhs \geq 0 \).
- `a` array of size at least \( \max(1, lda*n) \).
  - The array \( a \) contains the block diagonal matrix \( D \) and the multipliers used to obtain the factor \( U \) or \( L \) as computed by ?hetrf.
- `i1` index of the first row to swap.
- `i2` index of the second row to swap.

**Output Parameters**

- `a` if \( \text{info} = 0 \), the inverse of the original matrix.
  - If \( \text{info} = 'U' \), the upper triangular part of the inverse is formed and the part of \( A \) below the diagonal is not referenced.
If \( \text{info} = 'L' \), the lower triangular part of the inverse is formed and the part of \( A \) above the diagonal is not referenced.

**Return Values**

This function returns a value \( \text{info} \).

- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} = -i \), the \( i \)-th parameter had an illegal value.
- If \( \text{info} = -1011 \), memory allocation error occurred.

**See Also**

?hetrf

**?sfrk**

*Performs a symmetric rank-k operation for matrix in RFP format.*

**Syntax**

```c
lapack_int LAPACKE_ssfrk (int matrix_layout , char transr , char uplo , char trans ,
    lapack_int n , lapack_int k , float alpha , const float * a , lapack_int lda , float 
    beta , float * c);

lapack_int LAPACKE_dsfrk (int matrix_layout , char transr , char uplo , char trans ,
    lapack_int n , lapack_int k , double alpha , const double * a , lapack_int lda , double 
    beta , double * c);
```

**Include Files**

- mkl.h

**Description**

The ?sfrk routines perform a matrix-matrix operation using symmetric matrices. The operation is defined as

\[
\begin{align*}
C & := \alpha A A^T + \beta C, \\
\end{align*}
\]

or

\[
\begin{align*}
C & := \alpha A^T A + \beta C,
\end{align*}
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( C \) is an \( n \)-by-\( n \) symmetric matrix in rectangular full packed (RFP) format,
- \( A \) is an \( n \)-by-\( k \) matrix in the first case and a \( k \)-by-\( n \) matrix in the second case.

**Input Parameters**

- \( \text{matrix_layout} \) Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- \( \text{transr} \)

  - if \( \text{transr} = 'N' \) or \( 'n' \), the normal form of RFP \( C \) is stored;
  - if \( \text{transr} = 'T' \) or \( 't' \), the transpose form of RFP \( C \) is stored.
uplo

Specifies whether the upper or lower triangular part of the array \( c \) is used.

If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the array \( c \) is used.

If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the array \( c \) is used.

trans

Specifies the operation:

- If \( \text{trans} = 'N' \) or \( 'n' \), then \( C := \alpha A^T A + \beta C \);
- If \( \text{trans} = 'T' \) or \( 't' \), then \( C := \alpha A^T A + \beta C \);

\( n \)

Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

\( k \)

On entry with \( \text{trans} = 'N' \) or \( 'n' \), \( k \) specifies the number of columns of the matrix \( A \), and on entry with \( \text{trans} = 'T' \) or \( 't' \), \( k \) specifies the number of rows of the matrix \( A \).

The value of \( k \) must be at least zero.

alpha

Specifies the scalar \( \alpha \).

\( a \)

Array, size \( \max(1, \text{lda} \times \text{ka}) \), where \( \text{ka} \) is in the following table:

<table>
<thead>
<tr>
<th>Trans</th>
<th>Col_major</th>
<th>Row_major</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>( k )</td>
<td>( n )</td>
</tr>
<tr>
<td>'T'</td>
<td>( n )</td>
<td>( k )</td>
</tr>
</tbody>
</table>

Before entry with \( \text{trans} = 'N' \) or \( 'n' \), the leading \( n \)-by-\( k \) part of the array \( a \) must contain the matrix \( A \), otherwise the leading \( k \)-by-\( n \) part of the array \( a \) must contain the matrix \( A \).

lda

Specifies the leading dimension of \( a \) as declared in the calling (sub)program. \( \text{lda} \) is defined by the following table:

<table>
<thead>
<tr>
<th>Trans</th>
<th>Col_major</th>
<th>Row_major</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N'</td>
<td>( \max(1, n) )</td>
<td>( \max(1, k) )</td>
</tr>
<tr>
<td>'T'</td>
<td>( \max(1, k) )</td>
<td>( \max(1, n) )</td>
</tr>
</tbody>
</table>

beta

Specifies the scalar \( \beta \).

\( c \)

Array, size \( (n \times (n+1)/2) \). Before entry contains the symmetric matrix \( C \) in RFP format.

Output Parameters

\( c \)

- If \( \text{trans} = 'N' \) or \( 'n' \), then \( c \) contains \( C := \alpha A^T A + \beta C \);
- If \( \text{trans} = 'T' \) or \( 't' \), then \( c \) contains \( C := \alpha A^T A + \beta C \);

Return Values

This function returns a value \( \text{info} \).

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} < 0 \), the \( i \)-th parameter had an illegal value.

If \( \text{info} = -1011 \), memory allocation error occurred.
?hfrk
Performs a Hermitian rank-k operation for matrix in RFP format.

Syntax
lapack_int LAPACKE_chfrk ( int matrix_layout, char transr, char uplo, char trans,
lapack_int n, lapack_int k, float alpha, const lapack_complex_float* a, lapack_int lda,
float beta, lapack_complex_float* c );

lapack_int LAPACKE_zhfrk ( int matrix_layout, char transr, char uplo, char trans,
lapack_int n, lapack_int k, double alpha, const lapack_complex_double* a, lapack_int lda,
double beta, lapack_complex_double* c );

Include Files
• mkl.h

Description
The ?hfrk routines perform a matrix-matrix operation using Hermitian matrices. The operation is defined as

\[ C := \alpha A^H A + \beta C, \]

or

\[ C := \alpha A A^H + \beta C, \]

where:
\( \alpha \) and \( \beta \) are real scalars,

\( C \) is an \( n \)-by-\( n \) Hermitian matrix in RFP format,

\( A \) is an \( n \)-by-\( k \) matrix in the first case and a \( k \)-by-\( n \) matrix in the second case.

Input Parameters

- matrix_layout
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- transr
  if \( \text{transr} = 'N' \) or \( 'n' \), the normal form of RFP \( C \) is stored;
  if \( \text{transr} = 'C' \) or \( 'c' \), the conjugate-transpose form of RFP \( C \) is stored.

- uplo
  Specifies whether the upper or lower triangular part of the array \( c \) is used.
  If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the array \( c \) is used.
  If \( \text{uplo} = 'L' \) or \( 'l' \), then the low triangular part of the array \( c \) is used.

- trans
  Specifies the operation:
  if \( \text{trans} = 'N' \) or \( 'n' \), then \( C := \alpha A^H A + \beta C \);
  if \( \text{trans} = 'C' \) or \( 'c' \), then \( C := \alpha A A^H + \beta C \).

- n
  Specifies the order of the matrix \( C \). The value of \( n \) must be at least zero.

- k
  On entry with \( \text{trans} = 'N' \) or \( 'n' \), \( k \) specifies the number of columns of the matrix \( a \), and on entry with \( \text{trans} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), \( k \) specifies the number of rows of the matrix \( a \).
The value of k must be at least zero.

Specifies the scalar alpha.

Array, size max(1, lda*ka), where ka is in the following table:

<table>
<thead>
<tr>
<th></th>
<th>Col_major</th>
<th>Row_major</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans = 'N'</td>
<td>k</td>
<td>n</td>
</tr>
<tr>
<td>trans = 'T'</td>
<td>n</td>
<td>k</td>
</tr>
</tbody>
</table>

Before entry with trans = 'N' or 'n', the leading n-by-k part of the array a must contain the matrix A, otherwise the leading k-by-n part of the array a must contain the matrix A.

Specifies the leading dimension of a as declared in the calling (sub)program. lda is defined by the following table:

<table>
<thead>
<tr>
<th></th>
<th>Col_major</th>
<th>Row_major</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans = 'N'</td>
<td>max(1, n)</td>
<td>max(1, k)</td>
</tr>
<tr>
<td>trans = 'T'</td>
<td>max(1, k)</td>
<td>max(1, n)</td>
</tr>
</tbody>
</table>

Specifies the scalar beta.

Array, size (n*(n+1)/2). Before entry contains the Hermitian matrix C in in RFP format.

Output Parameters

c

If trans = 'N' or 'n', then c contains C := alpha*A*A^H + beta*C;
if trans = 'C' or 'c', then c contains C := alpha*A^H*A + beta*C;

Return Values

This function returns a value info.

If info = 0, the execution is successful.
If info < 0, the i-th parameter had an illegal value.
If info = -1011, memory allocation error occurred.

?tfsm
Solves a matrix equation (one operand is a triangular matrix in RFP format).

Syntax

```c
lapack_int LAPACKE_stfsm (int matrix_layout, char transr, char side, char uplo, char trans, char diag, lapack_int m, lapack_int n, float alpha, const float * a, float * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_dtfsm (int matrix_layout, char transr, char side, char uplo, char trans, char diag, lapack_int m, lapack_int n, double alpha, const double * a, double * b, lapack_int ldb);
```
lapack_int LAPACKE_ctfsm (int matrix_layout, char transr, char side, char uplo, char trans, char diag, lapack_int m, lapack_int n, lapack_complex_float alpha, const lapack_complex_float * a, lapack_complex_float * b, lapack_int ldb);
lapack_int LAPACKE_ztfsm (int matrix_layout, char transr, char side, char uplo, char trans, char diag, lapack_int m, lapack_int n, lapack_complex_double alpha, const lapack_complex_double * a, lapack_complex_double * b, lapack_int ldb);

Include Files

• mkl.h

Description

The ?tfsm routines solve one of the following matrix equations:

\[ \text{op}(A) \times X = \alpha B, \]

or

\[ X \times \text{op}(A) = \alpha B, \]

where:

\( \alpha \) is a scalar,

\( X \) and \( B \) are \( m \)-by-\( n \) matrices,

\( A \) is a unit, or non-unit, upper or lower triangular matrix in rectangular full packed (RFP) format.

\( \text{op}(A) \) can be one of the following:

• \( \text{op}(A) = A \) or \( \text{op}(A) = A^T \) for real flavors
• \( \text{op}(A) = A \) or \( \text{op}(A) = A^H \) for complex flavors

The matrix \( B \) is overwritten by the solution matrix \( X \).

Input Parameters

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **transr**
  if \( \text{transr} \) = 'N' or 'n', the normal form of RFP \( A \) is stored;
  if \( \text{transr} \) = 'T' or 't', the transpose form of RFP \( A \) is stored;
  if \( \text{transr} \) = 'C' or 'c', the conjugate-transpose form of RFP \( A \) is stored.

- **side**
  Specifies whether \( \text{op}(A) \) appears on the left or right of \( X \) in the equation:
  if \( \text{side} \) = 'L' or 'l', then \( \text{op}(A) \times X = \alpha B \);
  if \( \text{side} \) = 'R' or 'r', then \( X \times \text{op}(A) = \alpha B \).

- **uplo**
  Specifies whether the RFP matrix \( A \) is upper or lower triangular:
  if \( \text{uplo} \) = 'U' or 'u', then the matrix is upper triangular;
  if \( \text{uplo} \) = 'L' or 'l', then the matrix is lower triangular.

- **trans**
  Specifies the form of \( \text{op}(A) \) used in the matrix multiplication:
  if \( \text{trans} \) = 'N' or 'n', then \( \text{op}(A) = A \);
  if \( \text{trans} \) = 'T' or 't', then \( \text{op}(A) = A^T \);
if \( \text{trans} = 'C' \) or \( 'c' \), then \( \text{op}(A) = \text{conjg}(A') \).

**diag**

Specifies whether the RFP matrix \( A \) is unit triangular:
- if \( \text{diag} = 'U' \) or \( 'u' \) then the matrix is unit triangular;
- if \( \text{diag} = 'N' \) or \( 'n' \), then the matrix is not unit triangular.

**m**

Specifies the number of rows of \( B \). The value of \( m \) must be at least zero.

**n**

Specifies the number of columns of \( B \). The value of \( n \) must be at least zero.

**alpha**

Specifies the scalar \( \alpha \).
- When \( \alpha \) is zero, then \( a \) is not referenced and \( b \) need not be set before entry.

**a**

Array, size \((n*(n+1)/2)\). Contains the matrix \( A \) in RFP format.

**b**

Array, size \( \max(1, \text{ldb} \times n) \) for column major and \( \max(1, \text{ldb} \times m) \) for row major.
- Before entry, the leading \( m \)-by-\( n \) part of the array \( b \) must contain the right-hand side matrix \( B \).

**ldb**

Specifies the leading dimension of \( b \) as declared in the calling (sub)program. The value of \( \text{ldb} \) must be at least \( \max(1, m) \) for column major and \( \max(1, n) \) for row major.

**Output Parameters**

**b**

Overwritten by the solution matrix \( X \).

**Return Values**

This function returns a value \( \text{info} \).
- If \( \text{info} = 0 \), the execution is successful.
- If \( \text{info} < 0 \), the \( i \)-th parameter had an illegal value.
- If \( \text{info} = -1011 \), memory allocation error occurred.

**tfttp**

Copies a triangular matrix from the rectangular full packed format (TF) to the standard packed format (TP).

**Syntax**

```c
lapack_int LAPACKE_stfttp (int matrix_layout, char transr, char uplo, lapack_int n, const float * arf, float * ap);
lapack_int LAPACKE_dtfttp (int matrix_layout, char transr, char uplo, lapack_int n, const double * arf, double * ap);
lapack_int LAPACKE_ctfttp (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_float * arf, lapack_complex_float * ap);
lapack_int LAPACKE_ztfttp (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_double * arf, lapack_complex_double * ap);
```
Include Files

• mkl.h

Description

The routine copies a triangular matrix \( A \) from the Rectangular Full Packed (RFP) format to the standard packed format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters

- \texttt{matrix\_layout} Specifies whether matrix storage layout is row major (\texttt{LAPACK\_ROW\_MAJOR}) or column major (\texttt{LAPACK\_COL\_MAJOR}).

- \texttt{transr} = 'N': \( arf \) is in the Normal format,
  = 'T': \( arf \) is in the Transpose format (for \texttt{stfttp} and \texttt{dtfttp}),
  = 'C': \( arf \) is in the Conjugate-transpose format (for \texttt{ctfttp} and \texttt{ztfttp}).

- \texttt{uplo} Specifies whether \( A \) is upper or lower triangular:
  = 'U': \( A \) is upper triangular,
  = 'L': \( A \) is lower triangular.

- \texttt{n} The order of the matrix \( A \). \( n \geq 0 \).

- \texttt{arf} Array, size at least \( \max (1, n \times (n+1)/2) \).
  On entry, the upper or lower triangular matrix \( A \) stored in the RFP format.

Output Parameters

- \texttt{ap} Array, size at least \( \max (1, n \times (n+1)/2) \).
  On exit, the upper or lower triangular matrix \( A \), packed columnwise in a linear array.

Return Values

This function returns a value \texttt{info}.

If \( \texttt{info} = 0 \), the execution is successful.

If \( \texttt{info} < 0 \), the \( i \)-th parameter had an illegal value.

If \( \texttt{info} = -1011 \), memory allocation error occurred.

\texttt{?tfttr}

Copies a triangular matrix from the rectangular full packed format (TF) to the standard full format (TR).

Syntax

\begin{verbatim}
lapack_int LAPACKE_stfttr (int matrix_layout , char transr , char uplo , lapack_int n , const float * arf , float * a , lapack_int lda );
lapack_int LAPACKE_dtfttr (int matrix_layout , char transr , char uplo , lapack_int n , const double * arf , double * a , lapack_int lda );
\end{verbatim}
lapack_int LAPACKE_ctfttr (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_float * arf, lapack_complex_float * a, lapack_int lda);
lapack_int LAPACKE_ztfttr (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_double * arf, lapack_complex_double * a, lapack_int lda);

Include Files
• mkl.h

Description
The routine copies a triangular matrix $A$ from the Rectangular Full Packed (RFP) format to the standard full format. For the description of the RFP format, see Matrix Storage Schemes.

Input Parameters
matrix_layout Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
transr = 'N': arf is in the Normal format,
= 'T': arf is in the Transpose format (for stfttr and dtfttr),
= 'C': arf is in the Conjugate-transpose format (for ctfttr and ztfttr).
uplo Specifies whether $A$ is upper or lower triangular:
= 'U': $A$ is upper triangular,
= 'L': $A$ is lower triangular.
n The order of the matrices arf and $a$. $n \geq 0$.
arf Array, size at least $\max(1, n*(n+1)/2)$.
on entry, the upper or lower triangular matrix $A$ stored in the RFP format.
lda The leading dimension of the array $a$. $lda \geq \max(1,n)$.

Output Parameters
a Array, size $\max(1,lda*n)$.
on exit, the triangular matrix $A$. If $uplo = 'U'$, the leading $n$-by-$n$ upper triangular part of the array $a$ contains the upper triangular matrix, and the strictly lower triangular part of $a$ is not referenced. If $uplo = 'L'$, the leading $n$-by-$n$ lower triangular part of the array $a$ contains the lower triangular matrix, and the strictly upper triangular part of $a$ is not referenced.

Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info < 0, the $i$-th parameter had an illegal value.
If info = -1011, memory allocation error occurred.
 Computes a QR factorization of a real or complex "triangular-pentagonal" matrix, which is composed of a triangular block and a pentagonal block, using the compact WY representation for Q.

Syntax

```c
lapack_int LAPACKE_stpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, float * a, lapack_int lda, float * b, lapack_int ldb, float * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_dtpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, double * a, lapack_int lda, double * b, lapack_int ldb, double * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_ctpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb, lapack_complex_float * t, lapack_int ldt);
```

```c
lapack_int LAPACKE_ztpqrt2 (int matrix_layout, lapack_int m, lapack_int n, lapack_int l, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb, lapack_complex_double * t, lapack_int ldt);
```

Include Files

- mkl.h

Description

The input matrix C is an \( (n+m) \times n \) matrix

\[
C = \begin{bmatrix}
A & \leftarrow & n \times n \text{ upper triangular} \\
B & \leftarrow & m \times n \text{ pentagonal}
\end{bmatrix}
\]

where A is an \( n \times n \) upper triangular matrix, and B is an \( m \times n \) pentagonal matrix consisting of an \( (m-1) \times n \) rectangular matrix \( B1 \) on top of an \( l \times n \) upper trapezoidal matrix \( B2 \):

\[
B = \begin{bmatrix}
B1 & \leftarrow & (m-1) \times n \text{ rectangular} \\
B2 & \leftarrow & l \times n \text{ upper trapezoidal}
\end{bmatrix}
\]

The upper trapezoidal matrix \( B2 \) consists of the first \( l \) rows of an \( n \times n \) upper triangular matrix, where \( 0 \leq l \leq \min(m,n) \). If \( l=0 \), B is an \( m \times n \) rectangular matrix. If \( m=l=n \), B is upper triangular. The matrix W contains the elementary reflectors \( H(i) \) in the \( i \)th column below the diagonal (of A) in the \( (n+m) \times n \) input matrix C so that W can be represented as

\[
W = \begin{bmatrix}
I & \leftarrow & n \times n \text{ identity} \\
V & \leftarrow & m \times n \text{ pentagonal}
\end{bmatrix}
\]

Thus, V contains all of the information needed for W, and is returned in array b.
NOTE
V has the same form as B:

\[ V = \begin{bmatrix} V1 & \gets (m-l) \times n \text{ rectangular} \\ V2 & \gets l \times n \text{ upper trapezoidal} \end{bmatrix} \]

The columns of V represent the vectors which define the H(i) s.

The \((m+n)\)-by-\((m+n)\) block reflector \(H\) is then given by

\[ H = I - W^* T^* W \]

for real flavors, and

\[ H = I - W^* T^* W^H \]

for complex flavors

where \(W^T\) is the transpose of \(W\), \(W^H\) is the conjugate transpose of \(W\), and \(T\) is the upper triangular factor of the block reflector.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (`LAPACK_ROW_MAJOR`) or column major (`LAPACK_COL_MAJOR`).

- **m**
  The total number of rows in the matrix \(B\) \((m \geq 0)\).

- **n**
  The number of columns in \(B\) and the order of the triangular matrix \(A\) \((n \geq 0)\).

- **l**
  The number of rows of the upper trapezoidal part of \(B\) \((\min(m, n) \geq l \geq 0)\).

- **a, b**
  Arrays: \(a\), size `max(1, lda * n)` contains the \(n\)-by-\(n\) upper triangular matrix \(A\).

  \(b\), size `max(1, ldb * n)` for column major and `max(1, ldt * m)` for row major, the pentagonal \(m\)-by-\(n\) matrix \(B\). The first \((m-1)\) rows contain the rectangular \(B1\) matrix, and the next \(l\) rows contain the upper trapezoidal \(B2\) matrix.

- **lda**
  The leading dimension of \(a\); at least \(max(1, n)\).

- **ldb**
  The leading dimension of \(b\); at least \(max(1, m)\) for column major and \(max(1, n)\) for row major.

- **ldt**
  The leading dimension of \(t\); at least \(max(1, n)\).

**Output Parameters**

- **a**
  The elements on and above the diagonal of the array contain the upper triangular matrix \(R\).

- **b**
  The pentagonal matrix \(V\).

- **t**
  Array, size `max(1, ldt * n)`.

  The upper \(n\)-by-\(n\) upper triangular factor \(T\) of the block reflector.

**Return Values**

This function returns a value `info`. 

1206
If $info = 0$, the execution is successful.

If $info < 0$ and $info = -i$, the $i$th argument had an illegal value.

If $info = -1011$, memory allocation error occurred.

**?tprfb**

**Applies a real or complex "triangular-pentagonal"**

blocked reflector to a real or complex matrix, which is composed of two blocks.

**Syntax**

```c
lapack_int LAPACKE_stprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const float * v, lapack_int ldv, const float * t, lapack_int ldt, float * a, lapack_int lda, float * b, lapack_int ldb);

lapack_int LAPACKE_dtprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const double * v, lapack_int ldv, const double * t, lapack_int ldt, double * a, lapack_int lda, double * b, lapack_int ldb);

lapack_int LAPACKE_ctprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const lapack_complex_float * v, lapack_int ldv, const lapack_complex_float * t, lapack_int ldt, lapack_complex_float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);

lapack_int LAPACKE_ztprfb (int matrix_layout, char side, char trans, char direct, char storev, lapack_int m, lapack_int n, lapack_int k, lapack_int l, const lapack_complex_double * v, lapack_int ldv, const lapack_complex_double * t, lapack_int ldt, lapack_complex_double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);
```

**Include Files**

- mkl.h

**Description**

The ?tprfb routine applies a real or complex "triangular-pentagonal" block reflector $H$, $H^\top$, or $H^H$ from either the left or the right to a real or complex matrix $C$, which is composed of two blocks $A$ and $B$.

The block $B$ is $m$-by-$n$. If $side = 'R'$, $A$ is $m$-by-$k$, and if $side = 'L'$, $A$ is of size $k$-by-$n$.

\[
\begin{align*}
\text{side} = 'R' & \quad C = \begin{bmatrix} A & B \end{bmatrix} & C = \begin{bmatrix} B & A \end{bmatrix} \\
\text{side} = 'L' & \quad C = \begin{bmatrix} A \\ B \end{bmatrix} & C = \begin{bmatrix} B \\ A \end{bmatrix}
\end{align*}
\]

The pentagonal matrix $V$ is composed of a rectangular block $V_1$ and a trapezoidal block $V_2$. The size of the trapezoidal block is determined by the parameter $l$, where $0 \leq l \leq k$. If $l = k$, the $V_2$ block of $V$ is triangular; if $l = 0$, there is no trapezoidal block, thus $V = V_1$ is rectangular.
### Input Parameters

<table>
<thead>
<tr>
<th>matrix_layout</th>
<th>Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).</th>
</tr>
</thead>
</table>
| side | = 'L': apply $H$, $H^T$, or $H^H$ from the left,  
= 'R': apply $H$, $H^T$, or $H^H$ from the right. |
| trans | = 'N': apply $H$ (no transpose),  
= 'T': apply $H^T$ (transpose),  
= 'C': apply $H^H$ (conjugate transpose). |
| direct | Indicates how $H$ is formed from a product of elementary reflectors:  
= 'F': $H = H(1) H(2) \ldots H(k)$ (Forward),  
= 'B': $H = H(k) \ldots H(2) H(1)$ (Backward). |
| storev | Indicates how the vectors that define the elementary reflectors are stored:  
= 'C': Columns,  
= 'R': Rows. |
| $m$ | The total number of rows in the matrix $B$ ($m \geq 0$). |
| $n$ | The number of columns in $B$ ($n \geq 0$). |
| $k$ | The order of the matrix $T$, which is the number of elementary reflectors whose product defines the block reflector. ($k \geq 0$) |
| $l$ | The order of the trapezoidal part of $V$. ($k \geq l \geq 0$). |
An array containing the pentagonal matrix \( V \) (the elementary reflectors \( H(1), H(2), \ldots, H(k) \)). The size limitations depend on values of parameters \( \text{storev} \) and \( \text{side} \) as described in the following table:

<table>
<thead>
<tr>
<th>( \text{storev} = C )</th>
<th>( \text{storev} = R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{side} = L )</td>
<td>( \text{side} = R )</td>
</tr>
<tr>
<td>( \text{side} = L )</td>
<td>( \text{side} = R )</td>
</tr>
<tr>
<td>Column major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{ldv}*k) )</td>
<td>( \max(1, \text{ldv}*m) )</td>
</tr>
<tr>
<td>Row major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{ldv}*n) )</td>
<td>( \max(1, \text{ldv}*k) )</td>
</tr>
</tbody>
</table>

The leading dimension of the array \( v \). It should satisfy the following conditions:

<table>
<thead>
<tr>
<th>( \text{storev} = C )</th>
<th>( \text{storev} = R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{side} = L )</td>
<td>( \text{side} = R )</td>
</tr>
<tr>
<td>( \text{side} = L )</td>
<td>( \text{side} = R )</td>
</tr>
<tr>
<td>Column major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{ldv}*m) )</td>
<td>( \max(1, \text{ldv}*k) )</td>
</tr>
<tr>
<td>Row major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{ldv}*n) )</td>
<td>( \max(1, \text{ldv}*m) )</td>
</tr>
</tbody>
</table>

Array size \( \max(1, \text{ldt}*k) \). The triangular \( k \)-by-\( k \) matrix \( T \) in the representation of the block reflector.

The leading dimension of the array \( t \) (\( \text{ldt} \geq k \)).

The \( k \)-by-\( n \) or \( m \)-by-\( k \) matrix \( A \).

The leading dimension of the array \( a \) should satisfy the following conditions:

<table>
<thead>
<tr>
<th>( \text{side} = L )</th>
<th>( \text{side} = R )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{lda}*n) )</td>
<td>( \max(1, \text{lda}*k) )</td>
</tr>
<tr>
<td>Row major</td>
<td></td>
</tr>
<tr>
<td>( \max(1, \text{lda}*k) )</td>
<td>( \max(1, \text{lda}*m) )</td>
</tr>
</tbody>
</table>

Array size at least \( \max(1, \text{ldb}*n) \) for column major layout and \( \max(1, \text{ldb}*m) \) for row major layout, the \( m \)-by-\( n \) matrix \( B \).

The leading dimension of the array \( b \) (\( \text{ldb} \geq \max(1, m) \) for column major layout and \( \text{ldb} \geq \max(1, n) \) for row major layout).
Output Parameters

- **a**
  
  Contains the corresponding block of \( H^*C, H^*H, H^*H^, C^*H, C^*H^, \) or \( C^*H^ \).

- **b**
  
  Contains the corresponding block of \( H^*C, H^*H, H^*H^, C^*H, C^*H^, \) or \( C^*H^ \).

Return Values

This function returns a value `info`.

- **If** `info = 0`, the execution is successful.
- **If** `info < 0`, the i-th parameter had an illegal value.
- **If** `info = -1011`, memory allocation error occurred.

**?tpttf**

*Copies a triangular matrix from the standard packed format (TP) to the rectangular full packed format (TF).*

**Syntax**

```c
lapack_int LAPACKE_stpttf (int matrix_layout, char transr, char uplo, lapack_int n, const float * ap, float * arf);
lapack_int LAPACKE_dtpttf (int matrix_layout, char transr, char uplo, lapack_int n, const double * ap, double * arf);
lapack_int LAPACKE_ctpttf (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_float * ap, lapack_complex_float * arf);
lapack_int LAPACKE_ztpttf (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_double * ap, lapack_complex_double * arf);
```

**Include Files**

- mkl.h

**Description**

The routine copies a triangular matrix \( A \) from the standard packed format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

**Input Parameters**

- **matrix_layout**
  
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **transr**
  
  = 'N': \( arf \) must be in the Normal format,
  
  = 'T': \( arf \) must be in the Transpose format (for stpttf and dtpttf),
  
  = 'C': \( arf \) must be in the Conjugate-transpose format (for ctpttf and ztpttf).

- **uplo**
  
  Specifies whether \( A \) is upper or lower triangular:
  
  = 'U': \( A \) is upper triangular,
  
  = 'L': \( A \) is lower triangular.
The order of the matrix $A$. $n \geq 0$.

$ap$

Array, size at least max $(1, n*(n+1)/2)$.

On entry, the upper or lower triangular matrix $A$, packed in a linear array. See Matrix Storage Schemes for more information.

Output Parameters

$arf$

Array, size at least max $(1, n*(n+1)/2)$.

On exit, the upper or lower triangular matrix $A$ stored in the RFP format.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

$< 0$: if $info = -i$, the $i$-th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

?tpttr

Copies a triangular matrix from the standard packed format (TP) to the standard full format (TR).

Syntax

```c
lapack_int LAPACKE_stpttr (int matrix_layout, char uplo, lapack_int n, const float * ap, float * a, lapack_int lda);
lapack_int LAPACKE_dtpttr (int matrix_layout, char uplo, lapack_int n, const double * ap, double * a, lapack_int lda);
lapack_int LAPACKE_ctpttr (int matrix_layout, char uplo, lapack_int n, const lapack_complex_float * ap, lapack_complex_float * a, lapack_int lda);
lapack_int LAPACKE_ztpttr (int matrix_layout, char uplo, lapack_int n, const lapack_complex_double * ap, lapack_complex_double * a, lapack_int lda);
```

Include Files

- mkl.h

Description

The routine copies a triangular matrix $A$ from the standard packed format to the standard full format.

Input Parameters

$matrix_layout$

Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

$uplo$

Specifies whether $A$ is upper or lower triangular:

= 'U': $A$ is upper triangular,

= 'L': $A$ is lower triangular.
The order of the matrices \( a_{p} \) and \( a \). \( n \geq 0 \).

Array, size at least \( \max(1, n^*(n+1)/2) \). (see Matrix Storage Schemes).

The leading dimension of the array \( a \). \( lda \geq \max(1,n) \).

Output Parameters

Array, size \( \max(1,lda*n) \).

On exit, the triangular matrix \( A \). If \( uplo = \text{'U'} \), the leading \( n \)-by-\( n \) upper triangular part of the array \( a \) contains the upper triangular part of the matrix \( A \), and the strictly lower triangular part of \( a \) is not referenced. If \( uplo = \text{'L'} \), the leading \( n \)-by-\( n \) lower triangular part of the array \( a \) contains the lower triangular part of the matrix \( A \), and the strictly upper triangular part of \( a \) is not referenced.

Return Values

This function returns a value \( info \).

If \( info = 0 \), the execution is successful.

If \( info = -i \), the \( i \)-th parameter had an illegal value.

If \( info = -1011 \), memory allocation error occurred.

**?trttf**

Copies a triangular matrix from the standard full format (TR) to the rectangular full packed format (TF).

**Syntax**

```c
lapack_int LAPACKE_strttf (int matrix_layout, char transr, char uplo, lapack_int n, const float * a, lapack_int lda, float * arf);

lapack_int LAPACKE_dtrttf (int matrix_layout, char transr, char uplo, lapack_int n, const double * a, lapack_int lda, double * arf);

lapack_int LAPACKE_ctrttf (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_float * a, lapack_int lda, lapack_complex_float * arf);

lapack_int LAPACKE_ztrttf (int matrix_layout, char transr, char uplo, lapack_int n, const lapack_complex_double * a, lapack_int lda, lapack_complex_double * arf);
```

**Include Files**

- mkl.h

**Description**

The routine copies a triangular matrix \( A \) from the standard full format to the Rectangular Full Packed (RFP) format. For the description of the RFP format, see Matrix Storage Schemes.

**Input Parameters**

- `matrix_layout` Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).
transr

= 'N': arf must be in the Normal format,
= 'T': arf must be in the Transpose format (for strttf and dtrttf),
= 'C': arf must be in the Conjugate-transpose format (for ctrttf and ztrttf).

uplo

Specifies whether $A$ is upper or lower triangular:
= 'U': $A$ is upper triangular,
= 'L': $A$ is lower triangular.

n

The order of the matrix $A$. $n \geq 0$.

a

Array, size max(1,(lda*$n$)).

On entry, the triangular matrix $A$. If uplo = 'U', the leading $n$-by-$n$ upper triangular part of the array $a$ contains the upper triangular matrix, and the strictly lower triangular part of $a$ is not referenced. If uplo = 'L', the leading $n$-by-$n$ lower triangular part of the array $a$ contains the lower triangular matrix, and the strictly upper triangular part of $a$ is not referenced.

lda

The leading dimension of the array $a$. lda $\geq$ max(1,$n$).

Output Parameters

arf

Array, size at least max (1, $n*(n+1)/2$).

On exit, the upper or lower triangular matrix $A$ stored in the RFP format.

Return Values

This function returns a value info.

If info = 0, the execution is successful.

If info < 0, the $i$-th parameter had an illegal value.

If info = -1011, memory allocation error occurred.

?trttp

Copies a triangular matrix from the standard full format (TR) to the standard packed format (TP).

Syntax

lapack_int LAPACKE_strttp (int matrix_layout , char uplo , lapack_int n , const float * a , lapack_int lda , float * ap );

lapack_int LAPACKE_dtrttp (int matrix_layout , char uplo , lapack_int n , const double * a , lapack_int lda , double * ap );

lapack_int LAPACKE_ctrttp (int matrix_layout , char uplo , lapack_int n , const lapack_complex_float * a , lapack_int lda , lapack_complex_float * ap );

lapack_int LAPACKE_ztrttp (int matrix_layout , char uplo , lapack_int n , const lapack_complex_double * a , lapack_int lda , lapack_complex_double * ap );
Include Files

- mkl.h

Description

The routine copies a triangular matrix $A$ from the standard full format to the standard packed format.

Input Parameters

$uplo$

Specifies whether $A$ is upper or lower triangular:

- 'U': $A$ is upper triangular,
- 'L': $A$ is lower triangular.

$n$

The order of the matrix $A$, $n\geq 0$.

$a$

Array, size $\max(1, lda \times n)$.

On entry, the triangular matrix $A$. If $uplo = 'U'$, the leading $n$-by-$n$ upper triangular part of the array $a$ contains the upper triangular matrix, and the strictly lower triangular part of $a$ is not referenced. If $uplo = 'L'$, the leading $n$-by-$n$ lower triangular part of the array $a$ contains the lower triangular matrix, and the strictly upper triangular part of $a$ is not referenced.

$lda$

The leading dimension of the array $a$. $lda \geq \max(1, n)$.

Output Parameters

$ap$

Array, size at least $\max(1, n \times (n+1)/2)$.

On exit, the upper or lower triangular matrix $A$, packed columnwise in a linear array. (see Matrix Storage Schemes)

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info < 0$, the $i$-th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

?lacp2

Copies all or part of a real two-dimensional array to a complex array.

Syntax

```c
lapack_int LAPACKE_clacp2 (int matrix_layout, char uplo, lapack_int m, lapack_int n, const float * a, lapack_int lda, lapack_complex_float * b, lapack_int ldb);
```

```c
lapack_int LAPACKE_zlacp2 (int matrix_layout, char uplo, lapack_int m, lapack_int n, const double * a, lapack_int lda, lapack_complex_double * b, lapack_int ldb);
```

Include Files

- mkl.h
Description

The routine copies all or part of a real matrix $A$ to another matrix $B$.

Input Parameters

$\textit{matrix\_layout}$

Specifies whether matrix storage layout is row major (LAPACK_RowMajor) or column major (LAPACK_ColMajor).

$\textit{uplo}$

Specifies the part of the matrix $A$ to be copied to $B$.

- If $\textit{uplo} = 'U'$, the upper triangular part of $A$;
- If $\textit{uplo} = 'L'$, the lower triangular part of $A$.

Otherwise, all of the matrix $A$ is copied.

$m$

The number of rows in the matrix $A$ ($m \geq 0$).

$n$

The number of columns in $A$ ($n \geq 0$).

$a$

Array, size at least $\text{max}(1, lda \times n)$ for column major and $\text{max}(1, lda \times m)$ for row major, contains the $m$-by-$n$ matrix $A$.

- If $\textit{uplo} = 'U'$, only the upper triangle or trapezoid is accessed;
- if $\textit{uplo} = 'L'$, only the lower triangle or trapezoid is accessed.

$\textit{lda}$

The leading dimension of $a$; $\text{lda} \geq \text{max}(1, m)$ for column major and $\text{lda} \geq \text{max}(1, n)$ for row major.

$\textit{ldb}$

The leading dimension of the output array $b$; $\text{ldb} \geq \text{max}(1, m)$ for column major and $\text{ldb} \geq \text{max}(1, n)$ for row major.

Output Parameters

$b$

Array, size at least $\text{max}(1, ldb \times n)$ for column major layout and $\text{max}(1, ldb \times m)$ for row major layout, contains the $m$-by-$n$ matrix $B$.

On exit, $B = A$ in the locations specified by $\textit{uplo}$.

Return Values

This function returns a value $\textit{info}$.

- If $\textit{info} = 0$, the execution is successful.
- If $\textit{info} < 0$, the $i$-th parameter had an illegal value.
- If $\textit{info} = -1011$, memory allocation error occurred.

$\texttt{?larcm}$

Multiplies a square real matrix by a complex matrix.

Syntax

```c
lapack_int LAPACKE_clarcm(int \textit{matrix\_layout}, \textit{lapack_int} \textit{m}, \textit{lapack_int} \textit{n}, \textit{const float} * \textit{a}, \textit{lapack_int} \textit{lda}, \textit{const lapack_complex_float} * \textit{b}, \textit{lapack_int} \textit{ldb}, \textit{lapack_complex_float} * \textit{c}, \textit{lapack_int} \textit{ldc});
```
lapack_int LAPACKE_zlarcm(int matrix_layout, lapack_int m, lapack_int n, const double * a, lapack_int lda, const lapack_complex_double *b, lapack_int ldb, lapack_complex_double *c, lapack_int ldc);

**Description**

The routine performs a simple matrix-matrix multiplication of the form

\[ C = A \times B, \]

where \( A \) is \( m \)-by-\( m \) and real, \( B \) is \( m \)-by-\( n \) and complex, and \( C \) is \( m \)-by-\( n \) and complex.

**Input Parameters**

- **m**
  - The number of rows and columns of matrix \( A \) and the number of rows of matrix \( C \) (\( m \geq 0 \)).

- **n**
  - The number of columns of matrix \( B \) and the number of columns of matrix \( C \) (\( n \geq 0 \)).

- **a**
  - Array, size \([lda \times m]\). Contains the \( m \)-by-\( m \) matrix \( A \).

- **lda**
  - The leading dimension of the array \( a \), \( lda \geq \max(1, m) \).

- **b**
  - Array, size \((ldb, n)\). Contains the \( m \)-by-\( n \) matrix \( B \).

- **ldb**
  - The leading dimension of the array \( b \), \( ldb \geq \max(1, m) \) for column-major layout; \( ldb \geq \max(1, n) \) for row-major layout.

- **ldc**
  - The leading dimension of the array \( c \), \( ldc \geq \max(1, m) \) for column-major layout; \( ldc \geq \max(1, n) \) for row-major layout.

**Output Parameters**

- **c**
  - Array, size \((ldc, n)\). Contains the \( m \)-by-\( n \) matrix \( C \).

**Return Values**

This function returns a value \( info \). If \( info = 0 \), the execution is successful. If \( info = -i \), parameter \( i \) had an illegal value.

**mkl_？tpack**

Copies a triangular/symmetric matrix or submatrix from standard full format to standard packed format.

**Syntax**

```c
lapack_int LAPACKE_mkl_stppack (int matrix_layout, char uplo, char trans, lapack_int n, float* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const float* a, lapack_int lda);

lapack_int LAPACKE_mkl_dtppack (int matrix_layout, char uplo, char trans, lapack_int n, double* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const double* a, lapack_int lda);

lapack_int LAPACKE_mkl_ctppack (int matrix_layout, char uplo, char trans, lapack_int n, MKL_Complex8* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const MKL_Complex8* a, lapack_int lda);
```
lapack_int LAPACKE_mkl_ztppack (int matrix_layout, char uplo, char trans, lapack_int n, MKL_Complex16* ap, lapack_int i, lapack_int j, lapack_int rows, lapack_int cols, const MKL_Complex16* a, lapack_int lda);

Include Files

• mkl.h

Description

The routine copies a triangular or symmetric matrix or its submatrix from standard full format to packed format

\[ AP_{i:i+\text{rows}-1, j:j+\text{cols}-1} := \text{op}(A) \]

Standard packed formats include:

• TP: triangular packed storage
• SP: symmetric indefinite packed storage
• HP: Hermitian indefinite packed storage
• PP: symmetric or Hermitian positive definite packed storage

Full formats include:

• GE: general
• TR: triangular
• SY: symmetric indefinite
• HE: Hermitian indefinite
• PO: symmetric or Hermitian positive definite

NOTE

Any elements of the copied submatrix rectangular outside of the triangular part of the matrix \( AP \) are skipped.

Input Parameters

\( uplo \)

Specifies whether the matrix \( AP \) is upper or lower triangular.

If \( uplo = 'U' \), \( AP \) is upper triangular.

If \( uplo = 'L' \), \( AP \) is lower triangular.

\( trans \)

Specifies whether or not the copied block of \( A \) is transposed or not.

If \( trans = 'N', \) no transpose: \( \text{op}(A) = A \).

If \( trans = 'T', \) transpose: \( \text{op}(A) = A^T \).

If \( trans = 'C, \) conjugate transpose: \( \text{op}(A) = A^H \). For real data this is the same as \( trans = 'T' \).

\( n \)

The order of the matrix \( AP; \) \( n \geq 0 \)

\( i, j \)

Coordinates of the left upper corner of the destination submatrix in \( AP \).

If \( uplo='U', \) \( 1 \leq i \leq j \leq n \).

If \( uplo='L', \) \( 1 \leq j \leq i \leq n \).

\( rows \)

Number of rows in the destination submatrix. \( 0 \leq \text{rows} \leq n - i + 1 \).
Number of columns in the destination submatrix. $0 \leq cols \leq n - j + 1$.

Pointer to the source submatrix.

Array $a$ contains the rows-by-cols submatrix stored as unpacked rows-by-columns if $trans = 'N'$, or unpacked columns-by-rows if $trans = 'T'$ or $trans = 'C'$.

The size of $a$ is

<table>
<thead>
<tr>
<th>matrix_layout</th>
<th>lda=cols</th>
<th>lda=rows</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK_COL_MAJOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACK_ROW_MAJOR</td>
<td>lda=rows</td>
<td>lda=cols</td>
</tr>
</tbody>
</table>

**NOTE**

If there are elements outside of the triangular part of $AP$, they are skipped and are not copied from $a$.

The leading dimension of the array $a$.

<table>
<thead>
<tr>
<th>matrix_layout</th>
<th>lda=cols</th>
<th>lda=max(1, cols)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK_COL_MAJOR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LAPACK_ROW_MAJOR</td>
<td>lda=max(1, rows)</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

Array of size at least $\max(1, n(n+1)/2)$. The array $ap$ contains either the upper or the lower triangular part of the matrix $AP$ (as specified by $uplo$) in packed storage (see Matrix Storage Schemes). The submatrix of $ap$ from row $i$ to row $i + rows - 1$ and column $j$ to column $j + cols - 1$ is overwritten with a copy of the source matrix.

**Return Values**

This function returns a value $info$. If $info=0$, the execution is successful. If $info = -i$, the $i$-th parameter had an illegal value.

**mkl_?tpunpack**

Copies a triangular/symmetric matrix or submatrix from standard packed format to full format.
Description
The routine copies a triangular or symmetric matrix or its submatrix from standard packed format to full format.

\[ A := \text{op}(\text{AP}_{i:i+\text{rows}-1, j:j+\text{cols}-1}) \]

Standard packed formats include:
- TP: triangular packed storage
- SP: symmetric indefinite packed storage
- HP: Hermitian indefinite packed storage
- PP: symmetric or Hermitian positive definite packed storage

Full formats include:
- GE: general
- TR: triangular
- SY: symmetric indefinite
- HE: Hermitian indefinite
- PO: symmetric or Hermitian positive definite

**NOTE**
Any elements of the copied submatrix rectangular outside of the triangular part of AP are skipped.

Input Parameters

**uplo**

Specifies whether matrix AP is upper or lower triangular.
- If \( \text{uplo} = 'U' \), AP is upper triangular.
- If \( \text{uplo} = 'L' \), AP is lower triangular.

**trans**

Specifies whether or not the copied block of AP is transposed.
- If \( \text{trans} = 'N' \), no transpose: \( \text{op}(\text{AP}) = \text{AP} \).
If \( \text{trans} = 'T' \), transpose: \( \text{op}(\text{AP}) = \text{AP}^T \).

If \( \text{trans} = 'C' \), conjugate transpose: \( \text{op}(\text{AP}) = \text{AP}^H \). For real data this is the same as \( \text{trans} = 'T' \).

The order of the matrix \( \text{AP} \); \( n \geq 0 \).

Array, size at least \( \max(1, n(n+1)/2) \). The array \( \text{ap} \) contains either the upper or the lower triangular part of the matrix \( \text{AP} \) (as specified by \( \text{uplo} \)) in packed storage (see Matrix Storage Schemes). It is the source for the submatrix of \( \text{AP} \) from row \( i \) to row \( i + \text{rows} - 1 \) and column \( j \) to column \( j + \text{cols} - 1 \) to be copied.

Coordinates of left upper corner of the submatrix in \( \text{AP} \) to copy.

If \( \text{uplo} = 'U' \), \( 1 \leq i \leq j \leq n \).

If \( \text{uplo} = 'L' \), \( 1 \leq j \leq i \leq n \).

Number of rows to copy. \( 0 \leq \text{rows} \leq n - i + 1 \).

Number of columns to copy. \( 0 \leq \text{cols} \leq n - j + 1 \).

The leading dimension of array \( \text{a} \).

| matrix_layout = LAPACK_COL_MAJOR | \( \text{lda} \geq \max(1, \text{rows}) \) | \( \text{lda} \geq \max(1, \text{cols}) \) |
| matrix_layout = LAPACK_ROW_MAJOR | \( \text{lda} \geq \max(1, \text{cols}) \) | \( \text{lda} \geq \max(1, \text{rows}) \) |

Output Parameters

**a**

Pointer to the destination matrix. On exit, array \( \text{a} \) is overwritten with a copy of the unpacked \( \text{rows-by-cols} \) submatrix of \( \text{ap} \) unpacked rows-by-columns if \( \text{trans} = 'N' \), or unpacked columns-by-rows if \( \text{trans} = 'T' \) or \( \text{trans} = 'C' \).

The size of \( \text{a} \) is

| matrix_layout = LAPACK_COL_MAJOR | \( \text{lda} \times \text{cols} \) | \( \text{lda} \times \text{rows} \) |
| matrix_layout = LAPACK_ROW_MAJOR | \( \text{lda} \times \text{rows} \) | \( \text{lda} \times \text{cols} \) |

**NOTE**

If there are elements outside of the triangular part of \( \text{ap} \) indicated by \( \text{uplo} \), they are skipped and are not copied to \( \text{a} \).
Return Values
This function returns a value info. If info=0, the execution is successful. If info = -i, the i-th parameter had an illegal value.

LAPACK Utility Functions and Routines
This section describes LAPACK utility functions and routines.

See Also
lsame Tests two characters for equality regardless of the case.
lsamen Tests two character strings for equality regardless of the case.
second/dsecnd Returns elapsed time in seconds. Use to estimate real time between two calls to this function.
xerbla Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.

ilaver
Returns the version of the LAPACK library.

Syntax
void LAPACKE_ilaver (lapack_int * vers_major, lapack_int * vers_minor, lapack_int * vers_patch);

Include Files
• mkl.h

Description
This routine returns the version of the LAPACK library.

Output Parameters

vers_major
Returns the major version of the LAPACK library.

vers_minor
Returns the minor version from the major version of the LAPACK library.

vers_patch
Returns the patch version from the minor version of the LAPACK library.

?lamch
Determines machine parameters for floating-point arithmetic.

Syntax
float LAPACKE_slamch (char cmach);
double LAPACKE_dlamch (char cmach);

Include Files
• mkl.h

Description
The function \texttt{?lamch} determines single precision and double precision machine parameters.

**Input Parameters**

- \texttt{cmach}

  Specifies the value to be returned by \texttt{?lamch}:
  - \texttt{eps} = \texttt{e} or \texttt{E}, \texttt{val} = \texttt{eps}
  - \texttt{sfmin} = \texttt{s} or \texttt{s}, \texttt{val} = \texttt{sfmin}
  - \texttt{base} = \texttt{b} or \texttt{B}, \texttt{val} = \texttt{base}
  - \texttt{eps*base} = \texttt{p} or \texttt{P}, \texttt{val} = \texttt{eps*base}
  - \texttt{t} = \texttt{n} or \texttt{n}, \texttt{val} = \texttt{t}
  - \texttt{rnd} = \texttt{r} or \texttt{R}, \texttt{val} = \texttt{rnd}
  - \texttt{emin} = \texttt{m} or \texttt{M}, \texttt{val} = \texttt{emin}
  - \texttt{rmin} = \texttt{u} or \texttt{U}, \texttt{val} = \texttt{rmin}
  - \texttt{rmax} = \texttt{l} or \texttt{L}, \texttt{val} = \texttt{rmax}

  where
  - \texttt{eps} = relative machine precision;
  - \texttt{sfmin} = safe minimum, such that \(1/sfmin\) does not overflow;
  - \texttt{base} = base of the machine;
  - \texttt{prec} = \texttt{eps*base};
  - \texttt{t} = number of (base) digits in the mantissa;
  - \texttt{rnd} = 1.0 when rounding occurs in addition, 0.0 otherwise;
  - \texttt{emin} = minimum exponent before (gradual) underflow;
  - \texttt{rmin} = \texttt{underflow\_threshold} - \texttt{base}^{\texttt{emin}-1};
  - \texttt{emax} = largest exponent before overflow;
  - \texttt{rmax} = \texttt{overflow\_threshold} - (\texttt{base}^{\texttt{emax}})(1-\texttt{eps}).

**NOTE**

You can use a character string for \texttt{cmach} instead of a single character in order to make your code more readable. The first character of the string determines the value to be returned. For example, \texttt{'Precision'} is interpreted as \texttt{'p'}.

**Output Parameters**

- \texttt{val}

  Value returned by the function.

**LAPACK Test Functions and Routines**

This section describes LAPACK test functions and routines.
?lagge
Generates a general m-by-n matrix.

Syntax

```
lapack_int LAPACKE_slagge (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const float * d, float * a, lapack_int lda, lapack_int * iseed);
lapack_int LAPACKE_dlagge (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const double * d, double * a, lapack_int lda, lapack_int * iseed);
lapack_int LAPACKE_clagge (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const float * d, lapack_complex_float * a, lapack_int lda, lapack_int * iseed);
lapack_int LAPACKE_zlagge (int matrix_layout, lapack_int m, lapack_int n, lapack_int kl, lapack_int ku, const double * d, lapack_complex_double * a, lapack_int lda, lapack_int * iseed);
```

Include Files

- mkl.h

Description

The routine generates a general m-by-n matrix A, by pre- and post- multiplying a real diagonal matrix D with random matrices U and V:

\[ A := U \cdot D \cdot V, \]

where U and V are orthogonal for real flavors and unitary for complex flavors. The lower and upper bandwidths may then be reduced to kl and ku by additional orthogonal transformations.

Input Parameters

- \( m \)  
The number of rows of the matrix A (\( m \geq 0 \)).

- \( n \)  
The number of columns of the matrix A (\( n \geq 0 \)).

- \( kl \)  
The number of nonzero subdiagonals within the band of A (\( 0 \leq kl \leq m-1 \)).

- \( ku \)  
The number of nonzero superdiagonals within the band of A (\( 0 \leq ku \leq n-1 \)).

- \( d \)  
The array d with the dimension of \( \min(m, n) \) contains the diagonal elements of the diagonal matrix D.

- \( lda \)  
The leading dimension of the array a \( (lda \geq m) \) for column major layout and \( (lda \geq n) \) for row major layout.

- \( iseed \)  
The array iseed with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and iseed must be odd.
Output Parameters

- **a**
  The array $a$ with size at least $\max(1, lda*n)$ for column major layout and $\max(1, lda*m)$ for row major layout contains the generated $m$-by-$n$ matrix $A$.

- **iseed**
  The array $iseed$ contains the updated seed on exit.

Return Values

This function returns a value $info$.

If $info = 0$, the execution is successful.

If $info < 0$, the $i$-th parameter had an illegal value.

If $info = -1011$, memory allocation error occurred.

?!laghe

**Generates a complex Hermitian matrix.**

Syntax

```c
lapack_int LAPACKE_claghe (int matrix_layout, lapack_int n, lapack_int k, const float * d, lapack_complex_float * a, lapack_int lda, lapack_int * iseed);

lapack_int LAPACKE_zlaghe (int matrix_layout, lapack_int n, lapack_int k, const double * d, lapack_complex_double * a, lapack_int lda, lapack_int * iseed);
```

Include Files

- mkl.h

Description

The routine generates a complex Hermitian matrix $A$, by pre- and post- multiplying a real diagonal matrix $D$ with random unitary matrix:

$$A := U*D*U^\text{H}$$

The semi-bandwidth may then be reduced to $k$ by additional unitary transformations.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **n**
  The order of the matrix $A$ ($n \geq 0$).

- **k**
  The number of nonzero subdiagonals within the band of $A$ ($0 \leq k \leq n-1$).

- **d**
  The array $d$ with the dimension of $(n)$ contains the diagonal elements of the diagonal matrix $D$.

- **lda**
  The leading dimension of the array $a$ ($lda \geq n$).

- **iseed**
  The array $iseed$ with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and $iseed[3]$ must be odd.
Output Parameters

\(a\)

The array \(a\) of size at least \(\max(1, lda \times n)\) contains the generated \(n\)-by-\(n\) Hermitian matrix \(D\).

\(iseed\)

The array \(iseed\) contains the updated seed on exit.

Return Values

This function returns a value \(info\).

If \(info = 0\), the execution is successful.

If \(info < 0\), the \(i\)-th parameter had an illegal value.

If \(info = -1011\), memory allocation error occurred.

lagsy

Generates a symmetric matrix by pre- and post-multiplying a real diagonal matrix with a random unitary matrix.

Syntax

\[
\text{lapack_int LAPACKE_slagsy (int matrix_layout, lapack_int n, lapack_int k, const float * d, float * a, lapack_int lda, lapack_int * iseed);}\\
\text{lapack_int LAPACKE_dlagsy (int matrix_layout, lapack_int n, lapack_int k, const double * d, double * a, lapack_int lda, lapack_int * iseed);}\\
\text{lapack_int LAPACKE_clagsy (int matrix_layout, lapack_int n, lapack_int k, const float * d, lapack_complex_float * a, lapack_int lda, lapack_int * iseed);}\\
\text{lapack_int LAPACKE_zlagsy (int matrix_layout, lapack_int n, lapack_int k, const double * d, lapack_complex_double * a, lapack_int lda, lapack_int * iseed);}
\]

Include Files

- mkl.h

Description

The \(\text{lagsy}\) routine generates a symmetric matrix \(A\) by pre- and post-multiplying a real diagonal matrix \(D\) with a random matrix \(U\):

\[
A := U \ast D \ast U^T,
\]

where \(U\) is orthogonal for real flavors and unitary for complex flavors. The semi-bandwidth may then be reduced to \(k\) by additional unitary transformations.

Input Parameters

\(n\)

The order of the matrix \(A\) \((n \geq 0)\).

\(k\)

The number of nonzero subdiagonals within the band of \(A\) \((0 \leq k \leq n - 1)\).

\(d\)

The array \(d\) with the dimension of \(n\) contains the diagonal elements of the diagonal matrix \(D\).

\(lda\)

The leading dimension of the array \(a\) \((lda \geq n)\).
The array `iseed` with the dimension of 4 contains the seed of the random number generator. The elements must be between 0 and 4095 and `iseed[3]` must be odd.

Output Parameters

- **a**: The array of size \(\max(1, lda \times n)\) contains the generated symmetric \(n\)-by-\(n\) matrix \(D\).
- **iseed**: The array `iseed` contains the updated seed on exit.

Return Values

This function returns a value `info`.

- If `info = 0`, the execution is successful.
- If `info < 0`, the `i`-th parameter had an illegal value.
- If `info = -1011`, memory allocation error occurred.

### latms

*Generates a general \(m\)-by-\(n\) matrix with specific singular values.*

**Syntax**

```c
lapack_int LAPACKE_slatms (int matrix_layout, lapack_int m, lapack_int n, char dist, lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax, lapack_int kl, lapack_int ku, char pack, float * a, lapack_int lda);
```

```c
lapack_int LAPACKE_dlatms (int matrix_layout, lapack_int m, lapack_int n, char dist, lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax, lapack_int kl, lapack_int ku, char pack, double * a, lapack_int lda);
```

```c
lapack_int LAPACKE_clatms (int matrix_layout, lapack_int m, lapack_int n, char dist, lapack_int * iseed, char sym, float * d, lapack_int mode, float cond, float dmax, lapack_int kl, lapack_int ku, char pack, lapack_complex_float * a, lapack_int lda);
```

```c
lapack_int LAPACKE_zlatms (int matrix_layout, lapack_int m, lapack_int n, char dist, lapack_int * iseed, char sym, double * d, lapack_int mode, double cond, double dmax, lapack_int kl, lapack_int ku, char pack, lapack_complex_double * a, lapack_int lda);
```

**Include Files**

- mkl.h

**Description**

The `latms` routine generates random matrices with specified singular values, or symmetric/Hermitian matrices with specified eigenvalues for testing LAPACK programs.

It applies this sequence of operations:

1. Set the diagonal to \(d\), where \(d\) is input or computed according to `mode`, `cond`, `dmax`, and `sym` as described in Input Parameters.
2. Generate a matrix with the appropriate band structure, by one of two methods:
Method A

1. Generate a dense $m$-by-$n$ matrix by multiplying $d$ on the left and the right by random unitary matrices, then:
2. Reduce the bandwidth according to $k_l$ and $k_u$, using Householder transformations.

Method B:

Convert the bandwidth-0 (i.e., diagonal) matrix to a bandwidth-1 matrix using Givens rotations, "chasing" out-of-band elements back, much as in QR; then convert the bandwidth-1 to a bandwidth-2 matrix, etc.

Note that for reasonably small bandwidths (relative to $m$ and $n$) this requires less storage, as a dense matrix is not generated. Also, for symmetric or Hermitian matrices, only one triangle is generated.

Method A is chosen if the bandwidth is a large fraction of the order of the matrix, and $lda$ is at least $m$ (so a dense matrix can be stored.) Method B is chosen if the bandwidth is small (less than $(1/2)*n$ for symmetric or Hermitian or less than $0.3*n+m$ for nonsymmetric), or $lda$ is less than $m$ and not less than the bandwidth.

Pack the matrix if desired, using one of the methods specified by the pack parameter.

If Method B is chosen and band format is specified, then the matrix is generated in the band format and no repacking is necessary.

**Input Parameters**

- **matrix_layout**
  Specifies whether matrix storage layout is row major (LAPACK_ROW_MAJOR) or column major (LAPACK_COL_MAJOR).

- **m**
  The number of rows of the matrix $A$ ($m \geq 0$).

- **n**
  The number of columns of the matrix $A$ ($n \geq 0$).

- **dist**
  Specifies the type of distribution to be used to generate the random singular values or eigenvalues:
  - 'U': uniform distribution (0, 1)
  - 'S': symmetric uniform distribution (-1, 1)
  - 'N': normal distribution (0, 1)

- **iseed**
  Array with size 4.

  Specifies the seed of the random number generator. Values should lie between 0 and 4095 inclusive, and $iseed[3]$ should be odd. The random number generator uses a linear congruential sequence limited to small integers, and so should produce machine independent random numbers. The values of the array are modified, and can be used in the next call to ?latms to continue the same random number sequence.

- **sym**
  If $sym='S'$ or 'H', the generated matrix is symmetric or Hermitian, with eigenvalues specified by $d$, $cond$, $mode$, and $dmax$; they can be positive, negative, or zero.

  If $sym='P'$, the generated matrix is symmetric or Hermitian, with eigenvalues (which are singular, non-negative values) specified by $d$, $cond$, $mode$, and $dmax$. 

If `sym='N', the generated matrix is nonsymmetric, with singular, non-negative values specified by `d, `cond, `mode, and `dmax.

`d
Array, size `(MIN(`m, `n))
This array is used to specify the singular values or eigenvalues of `A (see the description of `sym). If `mode=0, then `d is assumed to contain the eigenvalues or singular values, otherwise elements of `d are computed according to `mode, `cond, and `dmax.

`mode
Describes how the singular/eigenvalues are specified.
- `mode = 0: use `d as input
- `mode = 1: set `d[0] = 1 and `d[1:`n - 1] = 1.0/`cond
- `mode = 2: set `d[0:`n - 2] = 1 and `d[`n - 1] = 1.0/`cond
- `mode = 3: set `d[i] = `cond/(i+1)
- `mode = 4: set `d[i] = 1 - i/(`n - 1)*((1 - 1/`cond)
- `mode = 5: set elements of `d to random numbers in the range (1/`cond, 1) such that their logarithms are uniformly distributed.
- `mode = 6: set elements of `d to random numbers from same distribution as the rest of the matrix.

`mode < 0 has the same meaning as ABS(`mode), except that the order of the elements of `d is reversed. Thus, if `mode is positive, `d has entries ranging from 1 to 1/`cond, if negative, from 1/`cond to 1.

If `sym='S' or 'H', and `mode is not 0, 6, nor -6, then the elements of `d are also given a random sign (multiplied by +1 or -1).

`cond
Used in setting `d as described for the `mode parameter. If used, `cond≥ 1.

`dmax
If `mode is not -6, 0 nor 6, the contents of `d, as computed according to `mode and `cond, are scaled by `dmax / max(abs(`d[i-1])); thus, the maximum absolute eigenvalue or singular value (the norm) is abs(`dmax).

**NOTE**
`dmax need not be positive: if `dmax is negative (or zero), `d will be scaled by a negative number (or zero).

`kl
Specifies the lower bandwidth of the matrix. For example, `kl=0 implies upper triangular, `kl=1 implies upper Hessenberg, and `kl being at least `m - 1 means that the matrix has full lower bandwidth. `kl must equal `ku if the matrix is symmetric or Hermitian.

`ku
Specifies the upper bandwidth of the matrix. For example, `ku=0 implies lower triangular, `ku=1 implies lower Hessenberg, and `ku being at least `n - 1 means that the matrix has full upper bandwidth. `kl must equal `ku if the matrix is symmetric or Hermitian.

`pack
Specifies packing of matrix:
- 'N': no packing
- 'U': zero out all subdiagonal entries (if symmetric or Hermitian)
- 'L': zero out all superdiagonal entries (if symmetric or Hermitian)
• 'B': store the lower triangle in band storage scheme (only if matrix symmetric, Hermitian, or lower triangular)
• 'Q': store the upper triangle in band storage scheme (only if matrix symmetric, Hermitian, or upper triangular)
• 'Z': store the entire matrix in band storage scheme (pivoting can be provided for by using this option to store $A$ in the trailing rows of the allocated storage)

Using these options, the various LAPACK packed and banded storage schemes can be obtained:

<table>
<thead>
<tr>
<th>Storage Type</th>
<th>'Z'</th>
<th>'B'</th>
<th>'Q'</th>
<th>'C'</th>
<th>'R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>GB: general band</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB: symmetric positive definite band</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SB: symmetric band</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HB: Hermitian band</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TB: triangular band</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PP: symmetric positive definite packed</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SP: symmetric packed</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HP: Hermitian packed</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>TP: triangular packed</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

If two calls to `?latms` differ only in the `pack` parameter, they generate mathematically equivalent matrices.

`lda` specifies the first dimension of $a$ as declared in the calling program.

If `pack='N', 'U', 'L', 'C', or 'R'`, then `lda` must be at least $m$ for column major or at least $n$ for row major.

If `pack='B' or 'Q'`, then `lda` must be at least $\text{MIN}(kl, m - 1)$ (which is equal to $\text{MIN}(ku, n - 1)$).

If `pack='Z'`, `lda` must be large enough to hold the packed array: $\text{MIN}( ku, n - 1) + \text{MIN}( kl, m - 1) + 1$.

**Output Parameters**

`iseed` contains the updated seed.

The array `d` contains the updated seed.

**NOTE**

The array `d` is not modified if `mode = 0`.

`a` is an array of size `lda` by $n$.

The array $a$ contains the generated $m$-by-$n$ matrix $A$. 
a is first generated in full (unpacked) form, and then packed, if so specified by pack. Thus, the first m elements of the first n columns are always modified. If pack specifies a packed or banded storage scheme, all lda elements of the first n columns are modified; the elements of the array which do not correspond to elements of the generated matrix are set to zero.

Return Values
This function returns a value info.
If info = 0, the execution is successful.
If info < 0, the i-th parameter had an illegal value.
If info = -1011, memory allocation error occurred.
If info = 2, cannot scale to dmax (maximum singular value is 0).
If info = 3, error return from lagge, ?laghe, or lagsy.

Additional LAPACK Routines (Included for Compatibility with Netlib LAPACK)

LAPACK_DECL lapack_int LAPACKE_chesv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_complex_float * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_float * b, lapack_int ldb);
LAPACK_DECL lapack_int LAPACKE_dsysv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, double * a, lapack_int lda, double * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2, double * b, lapack_int ldb);
LAPACK_DECL lapack_int LAPACKE_ssysv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, float * a, lapack_int lda, float * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2, float * b, lapack_int ldb);
LAPACK_DECL lapack_int LAPACKE_zhesv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * a, lapack_int lda, lapack_complex_double * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_double * b, lapack_int ldb);
LAPACK_DECL lapack_int LAPACKE_chetrf_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_complex_float * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2);
LAPACK_DECL lapack_int LAPACKE_dsytrf_aa_2stage (int matrix_layout, char uplo, lapack_int n, double * a, lapack_int lda, double * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2);
LAPACK_DECL lapack_int LAPACKE_ssytrf_aa_2stage (int matrix_layout, char uplo, lapack_int n, float * a, lapack_int lda, float * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2);
LAPACK_DECL lapack_int LAPACKE_zhetrf_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * tb, lapack_int ltb, lapack_int * ipiv, lapack_int * ipiv2);
LAPACK DECL lapack_int LAPACKE_chetrs_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_complex_float * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_float * b, lapack_int ldb);

LAPACK DECL lapack_int LAPACKE_dsytrs_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, double * a, lapack_int lda, double * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, double * b, lapack_int ldb);

LAPACK DECL lapack_int LAPACKE_ssytrs_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, float * a, lapack_int lda, float * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, float * b, lapack_int ldb);

LAPACK DECL lapack_int LAPACKE_zsysv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * a, lapack_int lda, lapack_complex_double * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_double * b, lapack_int ldb);

call csyev_aa_2stage (uplo, n, nrhs, a, lda, tb, ldb, ipiv, ipiv2, b, ldb, info);

LAPACK DECL lapack_int LAPACKE_csysv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_float * a, lapack_int lda, lapack_complex_float * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_float * b, lapack_int ldb);

call zsysv_aa_2stage (uplo, n, nrhs, a, lda, tb, ldb, ipiv, ipiv2, b, ldb, info);

LAPACK DECL lapack_int LAPACKE_zsysv_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_int nrhs, lapack_complex_double * a, lapack_int lda, lapack_complex_double * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2, lapack_complex_double * b, lapack_int ldb);

LAPACK DECL lapack_int LAPACKE_zsytrf_aa_2stage (int matrix_layout, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, lapack_complex_float * tb, lapack_int ldb, lapack_int * ipiv, lapack_int * ipiv2);
LAPACK_DECL lapack_int LAPACKE_ssyevr_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, float * a, lapack_int lda, float * vl, float vu, lapack_int il, lapack_int iu, float abstol, lapack_int * m, float * w, float * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_dsyevr_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, double * a, lapack_int lda, double * vl, double * vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_zheevx_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, float * a, lapack_int lda, float * vl, float * vu, lapack_int il, lapack_int iu, float abstol, lapack_int * m, float * w, float * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_ssygv_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, float * a, lapack_int lda, float * vl, float * vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_ssyevx_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, float * a, lapack_int lda, float * vl, float * vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_ssyevd_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, double * a, lapack_int lda, double * vl, double * vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_zheevd_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_sheev_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, float * a, lapack_int lda, float * vl, float * vu, lapack_int il, lapack_int iu, float abstol, lapack_int * m, float * w, float * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_cheev_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_float * a, lapack_int lda, float * vl, float * vu, lapack_int il, lapack_int iu, lapack_complex_float * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_zheev_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_zheevr_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_zheevd_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_zheevr_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_zheevd_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * ifail);

LAPACK_DECL lapack_int LAPACKE_zheevx_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, double * a, lapack_int lda, double * vl, double * vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * isuppz);

LAPACK_DECL lapack_int LAPACKE_zheevd_2stage (int matrix_layout, char jobz, char range, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda, lapack_complex_double * vl, lapack_complex_double * vu, lapack_int il, lapack_int iu, lapack_complex_double * abstol, lapack_int * m, lapack_int * w, lapack_int * z, lapack_int ldz, lapack_int * ifail);
LAPACK DECL lapack_int LAPACKE_zhegv_2stage (int matrix_layout, lapack_int itype, char jobz, char uplo, lapack_int n, lapack_complex_double * a, lapack_int lda,
lapack_complex_double * b, lapack_int ldb, double * w);

LAPACK DECL lapack_int LAPACKE_ssbevd_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, float * ab, lapack_int ldaB, float * w, float * z,
lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_ssbevx_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * w, double * z,
lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_zhegv_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, float * ab, lapack_int ldab, float * w, float * z,
lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_dsbevx_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * w, double * z,
lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_zhebv_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, char range, char uplo, lapack_int n, lapack_int kd, float * ab, lapack_int ldab, float * q,
lapack_int ldq, float vl, float vu, lapack_int il, lapack_int iu, float abstol,
lapack_int * m, float * w, float * z, lapack_int ldz, lapack_int * ifail);

LAPACK DECL lapack_int LAPACKE_zhebv_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, char range, char uplo, lapack_int n, lapack_int kd, double * ab, lapack_int ldab, double * q,
lapack_int ldq, double vl, double vu, lapack_int il, lapack_int iu, double abstol,
lapack_int * m, double * w, double * z, lapack_int ldz, lapack_int * ifail);

LAPACK DECL lapack_int LAPACKE_chbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, lapack_int kd, lapack_complex_float * ab, lapack_int ldab, float * w,
lapack_complex_float * z, lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_chbev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, char range, char uplo, lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab, float * w,
lapack_complex_double * z, lapack_int ldz);

LAPACK DECL lapack_int LAPACKE_zhebev_2stage (int matrix_layout, char jobz, char uplo,
lapack_int n, char range, char uplo, lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab, float * q,
lapack_complex_double * z, lapack_int ldz, lapack_int * ifail);

LAPACK DECL lapack_int LAPACKE_zhebev_2stage (int matrix_layout, char jobz, char range,
char uplo, lapack_int n, lapack_int kd, lapack_complex_double * ab, lapack_int ldab,
lapack_complex_double * q, lapack_int ldq, double vl, double vu, lapack_int il, lapack_int iu, double abstol, lapack_int * m, double * w, lapack_complex_double * z, lapack_int ldz, lapack_int * ifail);

For descriptions of these functions, please see http://www.netlib.org/lapack/explore-html/files.html.
ScaLAPACK Routines

Intel® Math Kernel Library implements routines from the ScaLAPACK package for distributed-memory architectures. Routines are supported for both real and complex dense and band matrices to perform the tasks of solving systems of linear equations, solving linear least-squares problems, eigenvalue and singular value problems, as well as performing a number of related computational tasks.

Intel® MKL ScaLAPACK routines are written in FORTRAN 77 with exception of a few utility routines written in C to exploit the IEEE arithmetic. All routines are available in all precision types: single precision, double precision, complex, and double complex precision. See the mkl_scalapack.h header file for C declarations of ScaLAPACK routines.

NOTE
ScaLAPACK routines are provided only for Intel® 64 or Intel® Many Integrated Core architectures.

See descriptions of ScaLAPACK computational routines that perform distinct computational tasks, as well as driver routines for solving standard types of problems in one call. Additionally, Intel® Math Kernel Library implements ScaLAPACK Auxiliary Routines, Utility Functions and Routines, and Matrix Redistribution/Copy Routines. The library includes routines for both real and complex data.

The <install_directory>/examples/scalapackf directory contains sample code demonstrating the use of ScaLAPACK routines.

Generally, ScaLAPACK runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of BLAS optimized for the target architecture. Intel® MKL version of ScaLAPACK is optimized for Intel® processors. For the detailed system and environment requirements, see Intel® MKL Release Notes and Intel® MKL Developer Guide.

For full reference on ScaLAPACK routines and related information, see [SLUG].

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Overview of ScaLAPACK Routines

The model of the computing environment for ScaLAPACK is represented as a one-dimensional array of processes (for operations on band or tridiagonal matrices) or also a two-dimensional process grid (for operations on dense matrices). To use ScaLAPACK, all global matrices or vectors should be distributed on this array or grid prior to calling the ScaLAPACK routines.

ScaLAPACK is closely tied to other components, including BLAS, BLACS, LAPACK, and PBLAS.
ScaLAPACK Array Descriptors

ScaLAPACK uses two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, and also allows use of BLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global matrix and its corresponding process and memory location is contained in the array called the array descriptor associated with each global matrix. The size of the array descriptor is denoted as $dlen_\_$. 

Let $A$ be a two-dimensional block cyclicly distributed matrix with the array descriptor array $desca$. The meaning of each array descriptor element depends on the type of the matrix $A$. The tables "Array descriptor for dense matrices" and "Array descriptor for narrow-band and tridiagonal matrices" describe the meaning of each element for the different types of matrices.

**Array descriptor for dense matrices ($dlen_\_=9$)**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Stored in</th>
<th>Description</th>
<th>Element Index Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>$dtype_a$</td>
<td>$desca[dtype_]$</td>
<td>Descriptor type ($=1$ for dense matrices).</td>
<td>0</td>
</tr>
<tr>
<td>$ctxt_a$</td>
<td>$desca[ctxt_]$</td>
<td>BLACS context handle for the process grid.</td>
<td>1</td>
</tr>
<tr>
<td>$m_a$</td>
<td>$desca[m_]$</td>
<td>Number of rows in the global matrix $A$.</td>
<td>2</td>
</tr>
<tr>
<td>$n_a$</td>
<td>$desca[n_]$</td>
<td>Number of columns in the global matrix $A$.</td>
<td>3</td>
</tr>
<tr>
<td>$mb_a$</td>
<td>$desca[mb_]$</td>
<td>Row blocking factor.</td>
<td>4</td>
</tr>
<tr>
<td>Element Name</td>
<td>Stored in</td>
<td>Description</td>
<td>Element Index Number</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>-------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>nb_a</td>
<td>desca[nb_]</td>
<td>Column blocking factor.</td>
<td>5</td>
</tr>
<tr>
<td>rsrca</td>
<td>desca[rsrca]</td>
<td>Process row over which the first row of the global matrix A is distributed.</td>
<td>6</td>
</tr>
<tr>
<td>csrca</td>
<td>desca[csrca]</td>
<td>Process column over which the first column of the global matrix A is distributed.</td>
<td>7</td>
</tr>
<tr>
<td>lllda</td>
<td>desca[lllda]</td>
<td>Leading dimension of the local matrix A.</td>
<td>8</td>
</tr>
</tbody>
</table>

**Array descriptor for narrow-band and tridiagonal matrices (dlen_ = 7)**

<table>
<thead>
<tr>
<th>Element Name</th>
<th>Stored in</th>
<th>Description</th>
<th>Element Index Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtype_a</td>
<td>desca[dtype_]</td>
<td>Descriptor type</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• dtype_a=501: 1-by-P grid,</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• dtype_a=502: P-by-1 grid.</td>
<td></td>
</tr>
<tr>
<td>ctxa</td>
<td>desca[ctxa]</td>
<td>BLACS context handle indicating the BLACS process grid over which the global matrix A is distributed. The context itself is global, but the handle (the integer value) can vary.</td>
<td>1</td>
</tr>
<tr>
<td>n_a</td>
<td>desca[n_a]</td>
<td>The size of the matrix dimension being distributed.</td>
<td>2</td>
</tr>
<tr>
<td>nb_a</td>
<td>desca[nb_a]</td>
<td>The blocking factor used to distribute the distributed dimension of the matrix A.</td>
<td>3</td>
</tr>
<tr>
<td>src_a</td>
<td>desca[src_a]</td>
<td>The process row or column over which the first row or column of the matrix A is distributed.</td>
<td>4</td>
</tr>
<tr>
<td>lllda</td>
<td>desca[lllda]</td>
<td>The leading dimension of the local matrix storing the local blocks of the distributed matrix A. The minimum value of lllda depends on dtype_a.</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• dtype_a=501: lllda ≥ max(size of undistributed dimension, 1),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• dtype_a=502: lllda ≥ max(nb_a, 1).</td>
<td></td>
</tr>
</tbody>
</table>

Not applicable: Reserved for future use. | 6 |

Similar notations are used for different matrices. For example: llb_a is the leading dimension of the local matrix storing the local blocks of the distributed matrix B and dtype_z is the type of the global matrix Z.

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by LOC_r() and LOC_c(), respectively. To compute these numbers, you can use the ScALAPACK tool routine numroc.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix sub(A) of the global matrix A defined by the following 6 values (for dense matrices):

- m The number of rows of sub(A)
- n The number of columns of sub(A)
- a A pointer to the local matrix containing the entire global matrix A
- ia The row index of sub(A) in the global matrix A
- ja The column index of sub(A) in the global matrix A
**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

---

**Naming Conventions for ScaLAPACK Routines**

For each routine introduced in this chapter, you can use the ScaLAPACK name. The naming convention for ScaLAPACK routines is similar to that used for LAPACK routines. A general rule is that each routine name in ScaLAPACK, which has an LAPACK equivalent, is simply the LAPACK name prefixed by initial letter p. **ScaLAPACK names** have the structure p?yyzzz or p?yyzz, which is described below.

The initial letter p is a distinctive prefix of ScaLAPACK routines and is present in each such routine.

The second symbol ? indicates the data type:

- **s**: real, single precision
- **d**: real, double precision
- **c**: complex, single precision
- **z**: complex, double precision

The second and third letters yy indicate the matrix type as:

- **ge**: general
- **gb**: general band
- **gg**: a pair of general matrices (for a generalized problem)
- **dt**: general tridiagonal (diagonally dominant-like)
- **db**: general band (diagonally dominant-like)
- **po**: symmetric or Hermitian positive-definite
- **pb**: symmetric or Hermitian positive-definite band
- **pt**: symmetric or Hermitian positive-definite tridiagonal
- **sy**: symmetric
- **st**: symmetric tridiagonal (real)
- **he**: Hermitian
- **or**: orthogonal
- **tr**: triangular (or quasi-triangular)
- **tz**: trapezoidal

The array descriptor for the global matrix A
For computational routines, the last three letters **zzz** indicate the computation performed and have the same meaning as for LAPACK routines.

For driver routines, the last two letters **zz** or three letters **zzz** have the following meaning:

- **sv**: a *simple* driver for solving a linear system
- **svx**: an *expert* driver for solving a linear system
- **ls**: a driver for solving a linear least squares problem
- **ev**: a simple driver for solving a symmetric eigenvalue problem
- **evd**: a simple driver for solving an eigenvalue problem using a divide and conquer algorithm
- **evx**: an expert driver for solving a symmetric eigenvalue problem
- **svd**: a driver for computing a singular value decomposition
- **gvx**: an expert driver for solving a generalized symmetric definite eigenvalue problem

*Simple* driver here means that the driver just solves the general problem, whereas an *expert* driver is more versatile and can also optionally perform some related computations (such, for example, as refining the solution and computing error bounds after the linear system is solved).

**ScaLAPACK Computational Routines**

In the sections that follow, the descriptions of ScaLAPACK computational routines are given. These routines perform distinct computational tasks that can be used for:

- Solving Systems of Linear Equations
- Orthogonal Factorizations and LLS Problems
- Symmetric Eigenproblems
- Nonsymmetric Eigenproblems
- Singular Value Decomposition
- Generalized Symmetric-Definite Eigenproblems

See also the respective driver routines.

**Systems of Linear Equations: ScaLAPACK Computational Routines**

ScaLAPACK supports routines for the systems of equations with the following types of matrices:

- general
- general banded
- general diagonally dominant-like banded (including general tridiagonal)
- symmetric or Hermitian positive-definite
- symmetric or Hermitian positive-definite banded
- symmetric or Hermitian positive-definite tridiagonal

A *diagonally dominant-like* matrix is defined as a matrix for which it is known in advance that pivoting is not required in the LU factorization of this matrix.

For the above matrix types, the library includes routines for performing the following computations: *factoring* the matrix; *equilibrating* the matrix; *solving* a system of linear equations; *estimating the condition number* of a matrix; *refining* the solution of linear equations and computing its error bounds; *inverting* the matrix. Note that for some of the listed matrix types only part of the computational routines are provided (for example, routines that refine the solution are not provided for band or tridiagonal matrices). See Table “Computational Routines for Systems of Linear Equations” for full list of available routines.
To solve a particular problem, you can either call two or more computational routines or call a corresponding driver routine that combines several tasks in one call. Thus, to solve a system of linear equations with a general matrix, you can first call `p?getrf` (LU factorization) and then `p?getrs` (computing the solution). Then, you might wish to call `p?gerfs` to refine the solution and get the error bounds. Alternatively, you can just use the driver routine `p?gesvx` which performs all these tasks in one call.

Table "Computational Routines for Systems of Linear Equations" lists the ScaLAPACK computational routines for factorizing, equilibrating, and inverting matrices, estimating their condition numbers, solving systems of equations with real matrices, refining the solution, and estimating its error.

**Computational Routines for Systems of Linear Equations**

<table>
<thead>
<tr>
<th>Matrix type, storage scheme</th>
<th>Factorize matrix</th>
<th>Equilibrate matrix</th>
<th>Solve system</th>
<th>Condition number</th>
<th>Estimate error</th>
<th>Invert matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>general band (partial pivoting)</td>
<td><code>p?gbtrf</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>general band (no pivoting)</td>
<td><code>p?dbtrf</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>general tridiagonal (no pivoting)</td>
<td><code>p?dttrf</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite, band</td>
<td><code>p?pbtrf</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>symmetric/Hermitian positive-definite, tridiagonal</td>
<td><code>p?pttrf</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>triangular</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this table, ? stands for `s` (single precision real), `d` (double precision real), `c` (single precision complex), or `z` (double precision complex).

**Matrix Factorization: ScaLAPACK Computational Routines**

This section describes the ScaLAPACK routines for matrix factorization. The following factorizations are supported:

- LU factorization of general matrices
- LU factorization of diagonally dominant-like matrices
- Cholesky factorization of real symmetric or complex Hermitian positive-definite matrices

You can compute the factorizations using full and band storage of matrices.

`p?getrf`

*Computes the LU factorization of a general m-by-n distributed matrix.*

**Syntax**

```c
void psgetrf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pdgetrf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pcgetrf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pzgetrf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
```
Include Files

- mkl_scalapack.h

Description

The `pgetrf` function forms the LU factorization of a general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(\text{ia} : \text{ia + m - 1}, \text{ja} : \text{ja + n - 1}) \) as

\[
A = P \cdot L \cdot U
\]

where \( P \) is a permutation matrix, \( L \) is lower triangular with unit diagonal elements (lower trapezoidal if \( m > n \)) and \( U \) is upper triangular (upper trapezoidal if \( m < n \)). \( L \) and \( U \) are stored in \( \text{sub}(A) \).

The function uses partial pivoting, with row interchanges.

**NOTE**

This function supports the Progress Routine feature. See `mkl_progress` for details.

Input Parameters

- \( m \) (global) The number of rows in the distributed matrix \( \text{sub}(A) \); \( m \geq 0 \).
- \( n \) (global) The number of columns in the distributed matrix \( \text{sub}(A) \); \( n \geq 0 \).
- \( a \) (local) Pointer into the local memory to an array of local size \( \text{lld}_a \cdot \text{LOCc}(\text{ja} + n - 1) \).
  Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.
- \( \text{ia}, \text{ja} \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.
- \( \text{desca} \) (global and local) array of size \( \text{dlen}_a \). The array descriptor for the distributed matrix \( A \).

Output Parameters

- \( a \) Overwritten by local pieces of the factors \( L \) and \( U \) from the factorization \( A = P \cdot L \cdot U \). The unit diagonal elements of \( L \) are not stored.
- \( \text{ipiv} \) (local) Array of size \( \text{LOCr}(\text{m}_a) + \text{mb}_a \).
  Contains the pivoting information: local row \( i \) was interchanged with global row \( \text{ipiv}[i-1] \). This array is tied to the distributed matrix \( A \).
- \( \text{info} \) (global)
  If \( \text{info} = 0 \), the execution is successful.
  \( \text{info} < 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i \times 100 + j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).
  If \( \text{info} = i > 0 \), \( u_{\text{ia}+i}, \text{ja}+j-1 \) is 0. The factorization has been completed, but the factor \( U \) is exactly singular. Division by zero will occur if you use the factor \( U \) for solving a system of linear equations.
See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?gbtrf**

Computes the LU factorization of a general n-by-n banded distributed matrix.

**Syntax**

```c
void psgbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, float *a, MKL_INT *ja,
MKL_INT *desca, MKL_INT *ipiv, float *af, MKL_INT *laf, float *work, MKL_INT
*lwork, MKL_INT *info);
void pdgbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, double *a, MKL_INT *ja,
MKL_INT *desca, MKL_INT *ipiv, double *af, MKL_INT *laf, double *work, MKL_INT
*lwork, MKL_INT *info);
void pcgbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_Complex8 *a, MKL_INT
*ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8
*work, MKL_INT *lwork, MKL_INT *info);
void pzgbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_Complex16 *a, MKL_INT
*ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16
*work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The **p?gbtrf** function computes the LU factorization of a general n-by-n real/complex banded distributed matrix \(A(1:n, ja:ja+n-1)\) using partial pivoting with row interchanges.

The resulting factorization is not the same factorization as returned from the LAPACK function ?gbtrf. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form

\[A(1:n, ja:ja+n-1) = P \cdot L \cdot U \cdot Q\]

where \(P\) and \(Q\) are permutation matrices, and \(L\) and \(U\) are banded lower and upper triangular matrices, respectively. The matrix \(Q\) represents reordering of columns for the sake of parallelism, while \(P\) represents reordering of rows for numerical stability using classic partial pivoting.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
**Input Parameters**

\( n \)  
(global) The number of rows and columns in the distributed submatrix \( A(1:n, ja:ja+n-1); n \geq 0 \).

\( bwl \)  
(global) The number of sub-diagonals within the band of \( A \)  
( \( 0 \leq bwl \leq n-1 \)).

\( bwu \)  
(global) The number of super-diagonals within the band of \( A \)  
( \( 0 \leq bwu \leq n-1 \)).

\( a \)  
(local)  
Pointer into the local memory to an array of local size \( lld_a \* \text{LOCc}(ja+n-1) \) where \( lld_a \geq 2 * bwl + 2 * bwu + 1 \).

Contains the local pieces of the \( n \)-by-\( n \) distributed banded matrix \( A(1:n, ja:ja+n-1) \) to be factored.

\( ja \)  
(global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( \text{desca} \)  
(global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( A \).

If \( dtype_a = 501 \), then \( dlen_ \geq 7 \); else if \( dtype_a = 1 \), then \( dlen_ \geq 9 \).

\( laf \)  
(local) The size of the array \( af \).

Must be \( laf \geq (nb_a+ bwu) * (bwu+bwl) + 6 * (bwu+bwl) * (bwl+2 * bwu) \).

If \( laf \) is not large enough, an error code will be returned and the minimum acceptable size will be returned in \( af[0] \).

\( \text{work} \)  
(local) Same type as \( a \). Workspace array of size \( lwork \).

\( lwork \)  
(local or global) The size of the \( \text{work} \) array (\( lwork \geq 1 \)). If \( lwork \) is too small, the minimal acceptable size will be returned in \( \text{work}[0] \) and an error code is returned.

**Output Parameters**

\( a \)  
On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by \( \text{LAPACK} \).

\( \text{ipiv} \)  
(local) array.

The size of \( \text{ipiv} \) must be \( \geq nb_a \).

Contains pivot indices for local factorizations. Note that you should not alter the contents of this array between factorization and solve.

\( af \)  
(local)  
Array of size \( laf \).
Auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( p?gbtrf \) and is stored in \( af \).

Note that if a linear system is to be solved using \( p?gbtrs \) after the factorization function, \( af \) must not be altered after the factorization.

On exit, \( work[0] \) contains the minimum value of \( lwork \) required.

\( info \) (global)

If \( info=0 \), the execution is successful.

\( info < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( info > 0 \):

If \( info = k \leq \text{NPROCS} \), the submatrix stored on processor \( info \) and factored locally was not nonsingular, and the factorization was not completed.

If \( info = k > \text{NPROCS} \), the submatrix stored on processor \( info-\text{NPROCS} \) representing interactions with other processors was not nonsingular, and the factorization was not completed.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

\( p?dbtrf \)

*Computes the LU factorization of a \( n \)-by-\( n \) diagonally dominant-like banded distributed matrix.*

**Syntax**

```c
void psdbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, float *a, MKL_INT *ja, MKL_INT *desca, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);
void pddbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, double *a, MKL_INT *ja, MKL_INT *desca, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);
void pcdbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzdbtrf (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The \( p?dbtrf \) function computes the LU factorization of a \( n \)-by-\( n \) real/complex diagonally dominant-like banded distributed matrix \( A(1:n, ja:ja+n-1) \) without pivoting.
**NOTE**
A matrix is called *diagonally dominant-like* if pivoting is not required for LU to be numerically stable.

Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

- **n**
  (global) The number of rows and columns in the distributed submatrix $A(1:n, ja:ja+n-1)$; $n \geq 0$.
- **bwl**
  (global) The number of sub-diagonals within the band of $A$
  $(0 \leq bwl \leq n-1)$.
- **bwu**
  (global) The number of super-diagonals within the band of $A$
  $(0 \leq bwu \leq n-1)$.
- **a**
  (local)
  Pointer into the local memory to an array of local size $lld_a \ast LOCc(ja+n-1)$.
  Contains the local pieces of the $n$-by-$n$ distributed banded matrix $A(1:n, ja:ja+n-1)$ to be factored.
- **ja**
  (global) The index in the global matrix $A$ indicating the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).
- **desca**
  (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.
  If $dtype_a = 501$, then $dlen_\geq 7$;
  else if $dtype_a = 1$, then $dlen_\geq 9$.
- **laf**
  (local) The size of the array $af$.
  Must be $laf_{\geq NB \ast (bwl+bwu)+6 \ast (\max(bwl,bwu))^2}$.
  If $laf$ is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.
- **work**
  (local) Workspace array of size $lwork$. 
lwork

(local or global) The size of the work array, must be \( lwork \geq (\max(bwl, bwu))^2 \). If \( lwork \) is too small, the minimal acceptable size will be returned in \( \text{work}[0] \) and an error code is returned.

Output Parameters

a

On exit, this array contains details of the factorization. Note that additional permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

af

(local)

Array of size \( \text{laf} \).

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( \text{p?dbtrf} \) and is stored in \( \text{af} \).

Note that if a linear system is to be solved using \( \text{p?dbtrs} \) after the factorization function, \( \text{af} \) must not be altered after the factorization.

work[0]

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

info

(global)

If \( info=0 \), the execution is successful.

\( info < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( info > 0 \):

If \( info = k \leq NPROCS \), the submatrix stored on processor \( info \) and factored locally was not diagonally dominant-like, and the factorization was not completed.

If \( info = k > NPROCS \), the submatrix stored on processor \( info-NPROCS \) representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dttrf

Computes the LU factorization of a diagonally dominant-like tridiagonal distributed matrix.

Syntax

```c
void psdttrf (MKL_INT *n, float *dl, float *d, float *du, MKL_INT *ja, MKL_INT *desca, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);
void pddttrf (MKL_INT *n, double *dl, double *d, double *du, MKL_INT *ja, MKL_INT *desca, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);
void pcdttrf (MKL_INT *n, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
```
void pzdttrf (MKL_INT *n, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?dttrf function computes the LU factorization of an \( n \times n \) real/complex diagonally dominant-like tridiagonal distributed matrix \( A(1:n, ja:ja+n-1) \) without pivoting for stability.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:
\[
A(1:n, ja:ja+n-1) = P*L*U*P^T,
\]
where \( P \) is a permutation matrix, and \( L \) and \( U \) are banded lower and upper triangular matrices, respectively.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\( n \)
(global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix \( A(1:n, ja:ja+n-1) \) \((n \geq 0)\).

\( dl, d, du \)
(local)
Pointers to the local arrays of size \( nb_a \) each.

On entry, the array \( dl \) contains the local part of the global vector storing the subdiagonal elements of the matrix. Globally, \( dl[0] \) is not referenced, and \( dl \) must be aligned with \( d \).

On entry, the array \( d \) contains the local part of the global vector storing the diagonal elements of the matrix.

On entry, the array \( du \) contains the local part of the global vector storing the super-diagonal elements of the matrix. \( du[n-1] \) is not referenced, and \( du \) must be aligned with \( d \).

\( ja \)
(global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( desca \)
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

If \( dtype_a = 501 \), then \( dlen \geq 7 \);
else if $dtype_a = 1$, then $dlen \geq 9$.

$laf$ (local) The size of the array $af$.

Must be $laf \geq 2*(NB+2)$.

If $laf$ is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.

$work$ (local) Same type as $d$. Workspace array of size $lwork$.

$lwork$ (local or global) The size of the $work$ array, must be at least $lwork \geq 8*NFCOL$.

Output Parameters

$dl, d, du$ On exit, overwritten by the information containing the factors of the matrix.

$af$ (local) Array of size $laf$.

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function $p?dttrf$ and is stored in $af$.

Note that if a linear system is to be solved using $p?dttrs$ after the factorization function, $af$ must not be altered.

$work[0]$ On exit, $work[0]$ contains the minimum value of $lwork$ required for optimum performance.

$info$ (global) If $info=0$, the execution is successful.

$info < 0$: If the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

$info > 0$: If $info = k \leq NPROCS$, the submatrix stored on processor $info$ and factored locally was not diagonally dominant-like, and the factorization was not completed.

If $info = k > NPROCS$, the submatrix stored on processor $info-NPROCS$ representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

$p?potrf$ Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite distributed matrix.

Syntax

```c
void pspotrf (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```
void pdpotrf (char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pcpotrf (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pzpotrf (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?potrf function computes the Cholesky factorization of a real symmetric or complex Hermitian positive-definite distributed n-by-n matrix A(ia:ia+n-1, ja:ja+n-1), denoted below as sub(A).

The factorization has the form
sub(A) = U^H*U if uplo='U', or
sub(A) = L*L^H if uplo='L'

where L is a lower triangular matrix and U is upper triangular.

Input Parameters

uplo (global) Indicates whether the upper or lower triangular part of sub(A) is stored. Must be 'U' or 'L'.
If uplo = 'U', the array a stores the upper triangular part of the matrix sub(A) that is factored as U^H*U.
If uplo = 'L', the array a stores the lower triangular part of the matrix sub(A) that is factored as L*L^H.

n (global) The order of the distributed matrix sub(A) (n≥0).

a (local) Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1). On entry, this array contains the local pieces of the n-by-n symmetric/Hermitian distributed matrix sub(A) to be factored. Depending on uplo, the array a contains either the upper or the lower triangular part of the matrix sub(A) (see uplo).

ia, ja (global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

Output Parameters

a The upper or lower triangular part of a is overwritten by the Cholesky factor U or L, as specified by uplo.
info (global) .
If $info=0$, the execution is successful; 

$info < 0$: if the $i$-th argument is an array, and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i\times100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$. 

If $info = k > 0$, the leading minor of order $k$, $A(ia:ia+k-1, ja:ja+k-1)$, is not positive-definite, and the factorization could not be completed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?pbtrf**
Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite banded distributed matrix.

**Syntax**
void pspbtrf (char *uplo, MKL_INT *n, MKL_INT *bw, float *a, MKL_INT *ja, MKL_INT *desca, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);
void pdpbtrf (char *uplo, MKL_INT *n, MKL_INT *bw, double *a, MKL_INT *ja, MKL_INT *desca, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);
void pcpbtrf (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzpbtrf (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

**Include Files**
- mkl_scalapack.h

**Description**
The p?pbtrf function computes the Cholesky factorization of an $n$-by-$n$ real symmetric or complex Hermitian positive-definite banded distributed matrix $A(1:n, ja:ja+n-1)$.

The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

$A(1:n, ja:ja+n-1) = P*U^H*U*P^T$, if uplo='U', or $A(1:n, ja:ja+n-1) = P*L^H*L*P^T$, if uplo='L',

where $P$ is a permutation matrix and $U$ and $L$ are banded upper and lower triangular matrices, respectively.
Input Parameters

**uplo**
(global) Must be 'U' or 'L'.
If uplo = 'U', upper triangle of A(1:n, ja:ja+n-1) is stored;
If uplo = 'L', lower triangle of A(1:n, ja:ja+n-1) is stored.

**n**
(global) The order of the distributed submatrix A(1:n, ja:ja+n-1).
(n≥0).

**bw**
(global) The number of superdiagonals of the distributed matrix if uplo = 'U', or
the number of subdiagonals if uplo = 'L' (bw≥0).

**a**
(local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1).
On entry, this array contains the local pieces of the upper or lower triangle
of the symmetric/Hermitian band distributed matrix A(1:n, ja:ja+n-1) to be factored.

**ja**
(global) The index in the global matrix A indicating the start of the matrix to be operated on (which may be either all of A or a submatrix of A).

**desca**
(global and local) array of size dlen_. The array descriptor for the
distributed matrix A.
If dtype_a = 501, then dlen_ ≥ 7;
else if dtype_a = 1, then dlen_ ≥ 9.

**laf**
(local) The size of the array af.
Must be laf ≥ (NB+2*bw)*bw.
If laf is not large enough, an error code will be returned and the minimum acceptable size will be returned in af[0].

**work**
(local) Workspace array of size lwork.

**lwork**
(local or global) The size of the work array, must be lwork≥bw^2.

Output Parameters

**a**
On exit, if info=0, contains the permuted triangular factor U or L from the
Cholesky factorization of the band matrix A(1:n, ja:ja+n-1), as specified by uplo.

**af**
(local)
Array of size \( laf \). Auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( \text{p}\text{pbtrf} \) and stored in \( af \). Note that if a linear system is to be solved using \( \text{p}\text{pbtrs} \) after the factorization function, \( af \) must not be altered.

\[
\text{work}[0]
\]

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\[
\text{info}
\]

(global)

If \( \text{info}=0 \), the execution is successful.

\[
\text{info}<0:
\]

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

\[
\text{info}>0:
\]

If \( \text{info} = k \leq \text{NPROCS} \), the submatrix stored on processor \( \text{info} \) and factored locally was not positive definite, and the factorization was not completed.

If \( \text{info} = k > \text{NPROCS} \), the submatrix stored on processor \( \text{info}-\text{NPROCS} \) representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{ppttrf}

\textit{Computes the Cholesky factorization of a symmetric (Hermitian) positive-definite tridiagonal distributed matrix.}

\textbf{Syntax}

\[
\begin{align*}
\text{void pspttrf} & \quad (\text{MKL_INT} \ast n, \text{float} \ast d, \text{float} \ast e, \text{MKL_INT} \ast ja, \text{MKL_INT} \ast \text{desca}, \text{float} \ast af, \text{MKL_INT} \ast laf, \text{float} \ast \text{work}, \text{MKL_INT} \ast lwork, \text{MKL_INT} \ast \text{info} ); \\
\text{void pdpttrf} & \quad (\text{MKL_INT} \ast n, \text{double} \ast d, \text{double} \ast e, \text{MKL_INT} \ast ja, \text{MKL_INT} \ast \text{desca}, \text{double} \ast af, \text{MKL_INT} \ast laf, \text{double} \ast \text{work}, \text{MKL_INT} \ast lwork, \text{MKL_INT} \ast \text{info} ); \\
\text{void pcpttrf} & \quad (\text{MKL_INT} \ast n, \text{float} \ast d, \text{MKL_Complex8} \ast e, \text{MKL_INT} \ast ja, \text{MKL_INT} \ast \text{desca}, \text{MKL_Complex8} \ast af, \text{MKL_INT} \ast laf, \text{MKL_Complex8} \ast \text{work}, \text{MKL_INT} \ast lwork, \text{MKL_INT} \ast \text{info} ); \\
\text{void pzpttrf} & \quad (\text{MKL_INT} \ast n, \text{double} \ast d, \text{MKL_Complex16} \ast e, \text{MKL_INT} \ast ja, \text{MKL_INT} \ast \text{desca}, \text{MKL_Complex16} \ast af, \text{MKL_INT} \ast laf, \text{MKL_Complex16} \ast \text{work}, \text{MKL_INT} \ast lwork, \text{MKL_INT} \ast \text{info} );
\end{align*}
\]

\textbf{Include Files}

- mkl_scalapack.h

\textbf{Description}

The \( \text{p}\text{pttrf} \) function computes the Cholesky factorization of an \( n \)-by-\( n \) real symmetric or complex hermitian positive-definite tridiagonal distributed matrix \( A(1:n, ja:ja+n-1) \).
The resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

The factorization has the form:

\[ A(1:n, ja:ja+n-1) = P*L*D*L^H*P^T, \] or

\[ A(1:n, ja:ja+n-1) = P*U^H*D*U*P^T, \]

where \( P \) is a permutation matrix, and \( U \) and \( L \) are tridiagonal upper and lower triangular matrices, respectively.

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Input Parameters

- **n** (global) The order of the distributed submatrix \( A(1:n, ja:ja+n-1) \) (\( n \geq 0 \)).

- **d, e** (local)

  Pointers into the local memory to arrays of size \( nb_a \) each.

  On entry, the array \( d \) contains the local part of the global vector storing the main diagonal of the distributed matrix \( A \).

  On entry, the array \( e \) contains the local part of the global vector storing the upper diagonal of the distributed matrix \( A \).

- **ja** (global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

- **desca** (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

  If \( dtype_a = 501 \), then \( dlen \geq 7 \);

  else if \( dtype_a = 1 \), then \( dlen \geq 9 \).

- **laf** (local) The size of the array \( af \).

  Must be \( laf \geq nb_a + 2 \).

  If \( laf \) is not large enough, an error code will be returned and the minimum acceptable size will be returned in \( af[0] \).

- **work** (local) Workspace array of size \( lwork \).

- **lwork** (local or global) The size of the \( work \) array, must be at least \( lwork \geq 8*NPCOL \).
Output Parameters

\( d, e \)

On exit, overwritten by the details of the factorization.

\( af \)

Array of size \( \text{laf} \).

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( \text{p?pttrf} \) and stored in \( af \).

Note that if a linear system is to be solved using \( \text{p?pttrs} \) after the factorization function, \( af \) must not be altered.

\( \text{work}[0] \)

On exit, \( \text{work}[0] \) contains the minimum value of \( \text{lwork} \) required for optimum performance.

\( \text{info} \)

If \( \text{info}=0 \), the execution is successful.

\( \text{info} < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

\( \text{info} > 0 \):

If \( \text{info} = k \leq \text{NPROCS} \), the submatrix stored on processor \( \text{info} \) and factored locally was not positive definite, and the factorization was not completed.

If \( \text{info} = k > \text{NPROCS} \), the submatrix stored on processor \( \text{info}-\text{NPROCS} \) representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Solving Systems of Linear Equations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for solving systems of linear equations. Before calling most of these routines, you need to factorize the matrix of your system of equations (see Routines for Matrix Factorization in this chapter). However, the factorization is not necessary if your system of equations has a triangular matrix.

\( \text{p?getrs} \)

Solves a system of distributed linear equations with a general square matrix, using the LU factorization computed by \( \text{p?getrf} \).

Syntax

void psgtrs (char *trans, MKL_INT *n, MKL_INT *nrhs, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);

void pdgetrs (char *trans, MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);
void pcgetrs (char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *info);

void pzgetrs (char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?getrs function solves a system of distributed linear equations with a general \( n \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(\text{id}_a:\text{id}_a+n-1, \text{id}_b:\text{id}_b+n-1) \) using the \( LU \) factorization computed by p?getrf.

The system has one of the following forms specified by \( \text{trans} \):

- \( \text{sub}(A) \times X = \text{sub}(B) \) (no transpose),
- \( \text{sub}(A)^{T} \times X = \text{sub}(B) \) (transpose),
- \( \text{sub}(A)^{H} \times X = \text{sub}(B) \) (conjugate transpose),

where \( \text{sub}(B) = B(\text{id}_b:ib+n-1, jb:jb+nrhs-1) \).

Before calling this function, you must call p?getrf to compute the \( LU \) factorization of \( \text{sub}(A) \).

Input Parameters

- \( \text{trans} \) (global) Must be 'N' or 'T' or 'C'.
  - Indicates the form of the equations:
    - If \( \text{trans} = 'N' \), then \( \text{sub}(A) \times X = \text{sub}(B) \) is solved for \( X \).
    - If \( \text{trans} = 'T' \), then \( \text{sub}(A)^{T} \times X = \text{sub}(B) \) is solved for \( X \).
    - If \( \text{trans} = 'C' \), then \( \text{sub}(A)^{H} \times X = \text{sub}(B) \) is solved for \( X \).

- \( n \) (global) The number of linear equations; the order of the matrix \( \text{sub}(A) \) \( (n \geq 0) \).

- \( nrhs \) (global) The number of right hand sides; the number of columns of the distributed matrix \( \text{sub}(B) \) \( (nrhs \geq 0) \).

- \( a, b \) (local)
  - Pointers into the local memory to arrays of local sizes \( lld_a \times \text{LOC}(ja+n-1) \) and \( lld_b \times \text{LOC}(jb+nrhs-1) \), respectively.
  - On entry, the array \( a \) contains the local pieces of the factors \( L \) and \( U \) from the factorization \( \text{sub}(A) = P \times L \times U \); the unit diagonal elements of \( L \) are not stored. On entry, the array \( b \) contains the right hand sides \( \text{sub}(B) \).

- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

- \( \text{desca} \) (global and local) array of size \( dlen_\_ \). The array descriptor for the distributed matrix \( A \).
**ipiv**

(local) Array of size of \( \text{LOCr}(m, a) + mb_a \). Contains the pivoting information: local row \( i \) of the matrix was interchanged with the global row \( ipiv[i-1] \).

This array is tied to the distributed matrix \( A \).

**ib, jb**

(global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the matrix \( \text{sub}(B) \), respectively.

**descb**

(global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( B \).

**Output Parameters**

**b**

On exit, overwritten by the solution distributed matrix \( X \).

**info**

If \( info=0 \), the execution is successful. \( info < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p\?gbtrs**

Solves a system of distributed linear equations with a general band matrix, using the LU factorization computed by \( p\?gbtrf \).

**Syntax**

```c
void psgbtrs (char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, float *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);

void pdgbtrs (char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, double *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);

void pcgbtrs (char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzgbtrs (char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h
Description
The \texttt{p?gbtrs} function solves a system of distributed linear equations with a general band distributed matrix \(\text{sub}(A) = A(1:n, ja:ja+n-1)\) using the \(LU\) factorization computed by \texttt{p?gbtrf}.

The system has one of the following forms specified by \texttt{trans}:
\[ \text{sub}(A) \cdot X = \text{sub}(B) \] (no transpose),
\[ \text{sub}(A)^T \cdot X = \text{sub}(B) \] (transpose),
\[ \text{sub}(A)^H \cdot X = \text{sub}(B) \] (conjugate transpose),
where \(\text{sub}(B) = B(ib:ib+n-1, 1:nrhs)\).

Before calling this function, you must call \texttt{p?gbtrf} to compute the \(LU\) factorization of \(\text{sub}(A)\).

Input Parameters

- \texttt{trans} (global) Must be 'N' or 'T' or 'C'.
  Indicates the form of the equations:
  - If \(\text{trans} = 'N'\), then \(\text{sub}(A) \cdot X = \text{sub}(B)\) is solved for \(X\).
  - If \(\text{trans} = 'T'\), then \(\text{sub}(A)^T \cdot X = \text{sub}(B)\) is solved for \(X\).
  - If \(\text{trans} = 'C'\), then \(\text{sub}(A)^H \cdot X = \text{sub}(B)\) is solved for \(X\).

- \(n\) (global) The number of linear equations; the order of the distributed matrix \(\text{sub}(A)\) \((n \geq 0)\).

- \(bwl\) (global) The number of sub-diagonals within the band of \(A\) \((0 \leq bwl \leq n-1)\).

- \(bwu\) (global) The number of super-diagonals within the band of \(A\) \((0 \leq bwu \leq n-1)\).

- \(nrhs\) (global) The number of right hand sides; the number of columns of the distributed matrix \(\text{sub}(B)\) \((nrhs \geq 0)\).

- \(a, b\) (local)
  Pointers into the local memory to arrays of local sizes \(lld_a \cdot \text{LOCc}(ja+n-1)\) and \(lld_b \cdot \text{LOCc}(nrhs)\), respectively.
  The array \(a\) contains details of the \(LU\) factorization of the distributed band matrix \(A\).
  On entry, the array \(b\) contains the local pieces of the right hand sides \(B(ib:ib+n-1, 1:nrhs)\).

- \(ja\) (global) The index in the global matrix \(A\) indicating the start of the matrix to be operated on (which may be either all of \(A\) or a submatrix of \(A\)).

- \(desca\) (global and local) array of size \(dlen_\_\). The array descriptor for the distributed matrix \(A\).
  - If \(\text{dtype}_a = 501\), then \(dlen \geq 7\);
  - else if \(\text{dtype}_a = 1\), then \(dlen \geq 9\).

- \(ib\) (global) The index in the global matrix \(A\) indicating the start of the matrix to be operated on (which may be either all of \(A\) or a submatrix of \(A\)).
descb  (global and local) array of size dlen_. The array descriptor for the
distributed matrix A.
If dtype_b = 502, then dlen ≥ 7;
else if dtype_b = 1, then dlen ≥ 9.

laf  (local) The size of the array af.
Must be laf≥nb_a*(bwl+bwu)+6*(bwl+bwu)*(bwl+2*bwu).
If laf is not large enough, an error code will be returned and the minimum
acceptable size will be returned in af[0].

work  (local) Same type as a. Workspace array of size lwork.

lwork  (local or global) The size of the work array, must be at least
lwork≥nrhs*(nb_a+2*bwl+4*bwu).

Output Parameters

ipiv  (local) array.
The size of ipiv must be ≥nb_a.
Contains pivot indices for local factorizations. Note that you should not alter
the contents of this array between factorization and solve.

b  On exit, overwritten by the local pieces of the solution distributed matrix X.

af  (local)
Array of size laf.
Auxiliary Fill-in space. The fill-in space is created in a call to the
factorization function p?gbtrf and is stored in af.
Note that if a linear system is to be solved using p?gbtrs after the
factorization function, af must not be altered after the factorization.

work[0]  On exit, work[0] contains the minimum value of lwork required for
optimum performance.

info  If info=0, the execution is successful.
info < 0:
If the i-th argument is an array and the j-th entry, indexed j - 1, had an
illegal value, then info = -(i*100+j); if the i-th argument is a scalar and
had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?dbtrs
Solves a system of linear equations with a diagonally
dominant-like banded distributed matrix using the
factorization computed by p?dbtrf.
Syntax

```c
void psdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
  float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
  float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );
void pddbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
  double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
  double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
void pcdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
  MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
  MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *
  lwork , MKL_INT *info );
void pzdbtrs (char *trans , MKL_INT *n , MKL_INT *bwl , MKL_INT *bwu , MKL_INT *nrhs ,
  MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib ,
  MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *
  lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The `p?dbtrs` function solves for \( X \) one of the systems of equations:

\[
\text{sub}(A)X = \text{sub}(B),
\]

\[
(\text{sub}(A))^T X = \text{sub}(B),
\]

or

\[
(\text{sub}(A))^H X = \text{sub}(B),
\]

where \( \text{sub}(A) = A(1:n, ja:ja+n-1) \) is a diagonally dominant-like banded distributed matrix, and \( \text{sub}(B) \) denotes the distributed matrix \( B(ib:ib+n-1, 1:nrhs) \).

This function uses the \( LU \) factorization computed by `p?dbtrf`.

Input Parameters

- **trans**
  - (global) Must be 'N' or 'T' or 'C'.
  - Indicates the form of the equations:
    - If `trans = 'N'`, then \( \text{sub}(A)X = \text{sub}(B) \) is solved for \( X \).
    - If `trans = 'T'`, then \( (\text{sub}(A))^T X = \text{sub}(B) \) is solved for \( X \).
    - If `trans = 'C'`, then \( (\text{sub}(A))^H X = \text{sub}(B) \) is solved for \( X \).

- **n**
  - (global) The order of the distributed matrix \( \text{sub}(A) \) (\( n \geq 0 \)).

- **bwl**
  - (global) The number of subdiagonals within the band of \( A \)
    - \( 0 \leq bwl \leq n-1 \).

- **bwu**
  - (global) The number of superdiagonals within the band of \( A \)
    - \( 0 \leq bwu \leq n-1 \).

- **nrhs**
  - (global) The number of right hand sides; the number of columns of the distributed matrix \( \text{sub}(B) \) (\( nrhs \geq 0 \)).
### a, b

Pointers into the local memory to arrays of local sizes \( lld_a \times LOCc(ja+n-1) \) and \( lld_b \times LOCc(nrhs) \), respectively.

On entry, the array \( a \) contains details of the LU factorization of the band matrix \( A \), as computed by \( p\text{?dbtrf} \).

On entry, the array \( b \) contains the local pieces of the right hand side distributed matrix sub\( (B) \).

### ja

The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

### desca

Array descriptor for the distributed matrix \( A \).

If \( dtype_a = 501 \), then \( dlen \geq 7 \); otherwise if \( dtype_a = 1 \), then \( dlen \geq 9 \).

### ib

The row index in the global matrix \( B \) indicating the first row of the matrix to be operated on (which may be either all of \( B \) or a submatrix of \( B \)).

### descb

Array descriptor for the distributed matrix \( B \).

If \( dtype_b = 502 \), then \( dlen \geq 7 \); otherwise if \( dtype_b = 1 \), then \( dlen \geq 9 \).

### af, work

Arrays of size \( laf \) and \( lwork \), respectively. The array \( af \) contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( p\text{?dbtrf} \) and is stored in \( af \).

The array \( work \) is a workspace array.

### laf

The size of the array \( af \).

Must be \( laf \geq NB \times (bwl+bwu)+6 \times (\max(bwl,bwu))^2 \).

If \( laf \) is not large enough, an error code will be returned and the minimum acceptable size will be returned in \( af[0] \).

### lwork

The size of the array \( work \), must be at least \( lwork \geq (\max(bwl,bwu))^2 \).

### Output Parameters

### b

On exit, this array contains the local pieces of the solution distributed matrix \( X \).

### work[0]

On exit, \( work[0] \) contains the minimum value of \( lwork \) required for optimum performance.

### info

If \( info=0 \), the execution is successful. \( info < 0 \):
If the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i\times100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?dttrs**
Solves a system of linear equations with a diagonally dominant-like tridiagonal distributed matrix using the factorization computed by p?dttrf.

**Syntax**

```c
void psdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *dl , float *d , float *du , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );
void pddttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , double *dl , double *d , double *du , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
void pcdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *dl , MKL_Complex8 *d , MKL_Complex8 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzdttrs (char *trans , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *dl , MKL_Complex16 *d , MKL_Complex16 *du , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**
The p?dttrs function solves for $X$ one of the systems of equations:

$\text{sub}(A)X = \text{sub}(B)$,

$(\text{sub}(A))^T X = \text{sub}(B)$, or

$(\text{sub}(A))^H X = \text{sub}(B)$,

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is a diagonally dominant-like tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses the LU factorization computed by p?dttrf.

**Input Parameters**

`trans` (global) Must be 'N' or 'T' or 'C'.

Indicates the form of the equations:

If $trans = 'N$', then $\text{sub}(A)X = \text{sub}(B)$ is solved for $X$.

If $trans = 'T$', then $(\text{sub}(A))^T X = \text{sub}(B)$ is solved for $X$. 

1260
If $trans = 'C'$, then $(\text{sub}(A))^H X = \text{sub}(B)$ is solved for $X$.

$n$

(global) The order of the distributed matrix $\text{sub}(A)$ ($n \geq 0$).

$nrhs$

(global) The number of right hand sides; the number of columns of the distributed matrix $\text{sub}(B)$ ($nrhs \geq 0$).

$dl, d, du$

(local)

Pointers to the local arrays of size $nb_a$ each.

On entry, these arrays contain details of the factorization. Globally, $dl[0]$ and $du[n-1]$ are not referenced; $dl$ and $du$ must be aligned with $d$.

$ja$

(global) The index in the global matrix $A$ indicating the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).

$desca$

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

If $\text{dtype}_a = 501$ or $\text{dtype}_a = 502$, then $dlen_\_ \geq 7$;

else if $\text{dtype}_a = 1$, then $dlen_\_ \geq 9$.

$b$

(local) Same type as $d$.

Pointer into the local memory to an array of local size $lld_b*LOCc(nrhs)$

On entry, the array $b$ contains the local pieces of the $n$-by-$nrhs$ right hand side distributed matrix $\text{sub}(B)$.

$ib$

(global) The row index in the global matrix $B$ indicating the first row of the matrix to be operated on (which may be either all of $B$ or a submatrix of $B$).

$descb$

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $B$.

If $\text{dtype}_b = 502$, then $dlen_\_ \geq 7$;

else if $\text{dtype}_b = 1$, then $dlen_\_ \geq 9$.

$af, work$

(local)

Arrays of size $laf$ and $lwork$, respectively.

The array $af$ contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function p\texttt{dttrf} and is stored in $af$. If a linear system is to be solved using p\texttt{dttrs} after the factorization function, $af$ must not be altered.

The array $work$ is a workspace array.

$laf$

(local) The size of the array $af$.

Must be $laf \geq NB*(bwl+bwu)+6*(bwl+bwu)* (bw1+2*bwu)$.

If $laf$ is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.

$lwork$

(local or global) The size of the array $work$, must be at least $lwork \geq 10*NPCOL+4*nrhs$. 
Output Parameters

- **b**: On exit, this array contains the local pieces of the solution distributed matrix X.
- **work[0]**: On exit, work[0] contains the minimum value of lwork required for optimum performance.
- **info**: If info=0, the execution is successful. info < 0:
  - If the i-th argument is an array and the j-th entry, indexed j-1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?potrs**
Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian distributed positive-definite matrix.

Syntax

```c
void pspotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pdpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pcpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
void pzpotrs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

The p?potrs function solves for X a system of distributed linear equations in the form:

$$\text{sub}(A)*X = \text{sub}(B)$$

where \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) is an \( n \)-by-\( n \) real symmetric or complex Hermitian positive definite distributed matrix, and \( \text{sub}(B) \) denotes the distributed matrix \( B(ib:ib+n-1, jb:jb+nrhs-1) \).

This function uses Cholesky factorization

- \( \text{sub}(A) = U^H*U \), or \( \text{sub}(A) = L*L^H \)

computed by p?potrf.
Input Parameters

**uplo**

(global) Must be 'U' or 'L'.

If **uplo** = 'U', upper triangle of sub($A$) is stored;

If **uplo** = 'L', lower triangle of sub($A$) is stored.

**n**

(global) The order of the distributed matrix sub($A$) ($n \geq 0$).

**nrhs**

(global) The number of right hand sides; the number of columns of the distributed matrix sub($B$) ($nrhs \geq 0$).

**a, b**

(local)

Pointers into the local memory to arrays of local sizes

$lidi_a*\text{LOCc}(ja+n-1)$ and $lidi_b*\text{LOCc}(jb+nrhs-1)$, respectively.

The array $a$ contains the factors $L$ or $U$ from the Cholesky factorization sub($A$) = $L*L^H$ or sub($A$) = $U^H*U$, as computed by $p?potrf$.

On entry, the array $b$ contains the local pieces of the right hand sides sub($B$).

**ia, ja**

(global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the matrix sub($A$), respectively.

**desca**

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

**ib, jb**

(global) The row and column indices in the global matrix $B$ indicating the first row and the first column of the matrix sub($B$), respectively.

**descb**

(local) array of size $dlen_\_$. The array descriptor for the distributed matrix $B$.

Output Parameters

**b**

Overwritten by the local pieces of the solution matrix $X$.

**info**

If info=0, the execution is successful.

info < 0: if the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then info = $-(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then info = $-i$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

$p?pbtrs$

Solves a system of linear equations with a Cholesky-factored symmetric/Hermitian positive-definite band matrix.

Syntax

```c
void pspbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb , float *af , MKL_INT *laf , float *work , MKL_INT *iwork , MKL_INT *info );
```
void pdpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb , double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );

void pcpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );

void pzpbtrs (char *uplo , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *descb , MKL_Complex16 *af , MKL_INT *laf , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

Include Files
• mkl_scalapack.h

Description
The p?pbtrs function solves for $X$ a system of distributed linear equations in the form:

$$\text{sub}(A)X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an $n$-by-$n$ real symmetric or complex Hermitian positive definite distributed band matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses Cholesky factorization

$$\text{sub}(A) = P*U^H*U*P^T,$$

computed by p?pbtrf.

Input Parameters

- **uplo**
  
  (global) Must be 'U' or 'L'.
  
  If uplo = 'U', upper triangle of sub$(A)$ is stored;
  
  If uplo = 'L', lower triangle of sub$(A)$ is stored.

- **n**
  
  (global) The order of the distributed matrix sub$(A)$ ($n \geq 0$).

- **bw**
  
  (global) The number of superdiagonals of the distributed matrix if uplo = 'U', or the number of subdiagonals if uplo = 'L' ($bw \geq 0$).

- **nrhs**
  
  (global) The number of right hand sides; the number of columns of the distributed matrix sub$(B)$ ($nrhs \geq 0$).

- **a, b**
  
  (local)
  
  Pointers into the local memory to arrays of local sizes $lld_a*LOCc(ja+n-1)$ and $lld_b*LOCc(nrhs-1)$, respectively.
  
  The array $a$ contains the permuted triangular factor $U$ or $L$ from the Cholesky factorization $\text{sub}(A) = P*U^H*U*P^T$, or $\text{sub}(A) = P*L^H*P^T$ of the band matrix $A$, as returned by p?pbtrf.
  
  On entry, the array $b$ contains the local pieces of the $n$-by-$nrhs$ right hand side distributed matrix sub$(B)$.

- **ja**
  
  (global) The index in the global matrix $A$ indicating the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).
(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

If $dtype\_a = 501$, then $dlen \geq 7$;
else if $dtype\_a = 1$, then $dlen \geq 9$.

$ib$ (global) The row index in the global matrix $B$ indicating the first row of the matrix sub($B$).

$descb$ (global and local) array of size $dlen\_$. The array descriptor for the distributed matrix $B$.

If $dtype\_b = 502$, then $dlen \geq 7$;
else if $dtype\_b = 1$, then $dlen \geq 9$.

$af$, $work$ (local) Arrays, same type as $a$.

The array $af$ is of size $laf$. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function $p\?dbtrf$ and is stored in $af$.

The array $work$ is a workspace array of size $lwork$.

$laf$ (local) The size of the array $af$.

Must be $laf\geq nrhs\times bw$.

If $laf$ is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.

$lwork$ (local or global) The size of the array $work$, must be at least $lwork\geq bw^2$.

**Output Parameters**

$b$ On exit, if $info=0$, this array contains the local pieces of the $n$-by-$nrhs$ solution distributed matrix $X$.

$work[0]$ On exit, $work[0]$ contains the minimum value of $lwork$ required for optimum performance.

$info$ If $info=0$, the execution is successful.

$info < 0$:

If the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i\times100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

$p\?pttrs$

Solves a system of linear equations with a symmetric (Hermitian) positive-definite tridiagonal distributed matrix using the factorization computed by $p\?pttrf$. 4
Syntax

```c
void pspttrs (MKL_INT *n, MKL_INT *nrhs, float *d, float *e, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);
void pdpttrs (MKL_INT *n, MKL_INT *nrhs, double *d, double *e, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);
void pcpttrs (char *uplo, MKL_INT *n, MKL_INT *nrhs, float *d, MKL_Complex8 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzpttrs (char *uplo, MKL_INT *n, MKL_INT *nrhs, double *d, MKL_Complex16 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

Include Files
- mkl_scalapack.h

Description
The `p?pttrs` function solves for $X$ a system of distributed linear equations in the form:

$$\text{sub}(A) \cdot X = \text{sub}(B),$$

where $\text{sub}(A) = A(1:n, ja:ja+n-1)$ is an $n$-by-$n$ real symmetric or complex Hermitian positive definite tridiagonal distributed matrix, and $\text{sub}(B)$ denotes the distributed matrix $B(ib:ib+n-1, 1:nrhs)$.

This function uses the factorization $\text{sub}(A) = P \cdot L \cdot D \cdot L^T \cdot P^T$, or $\text{sub}(A) = P \cdot U^H \cdot D \cdot U \cdot P^T$ computed by `p?pttrf`.

Input Parameters
- **uplo** (global, used in complex flavors only)
  Must be `'U'` or `'L'`.
  - If `uplo = 'U'`, upper triangle of sub$(A)$ is stored;
  - If `uplo = 'L'`, lower triangle of sub$(A)$ is stored.
- **n** (global) The order of the distributed matrix sub$(A)$ ($n \geq 0$).
- **nrhs** (global) The number of right hand sides; the number of columns of the distributed matrix sub$(B)$ ($nrhs \geq 0$).
- **d, e** (local)
  Pointers into the local memory to arrays of size $nb_a$ each.
  These arrays contain details of the factorization as returned by `p?pttrf`.
- **ja** (global) The index in the global matrix $A$ indicating the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).
desca

(global and local) array of size dlen_. The array descriptor for the distributed matrix A.

If dtype_a = 501 or dtype_a = 502, then dlen ≥ 7;
else if dtype_a = 1, then dlen ≥ 9.

b

(local) Same type as d, e.

Pointer into the local memory to an array of local size lld_b*LOCc(nrhs).

On entry, the array b contains the local pieces of the n-by-nrhs right hand side distributed matrix sub(B).

ib

(global) The row index in the global matrix B indicating the first row of the matrix to be operated on (which may be either all of B or a submatrix of B).

descb

(global and local) array of size dlen_. The array descriptor for the distributed matrix B.

If dtype_b = 502, then dlen ≥ 7;
else if dtype_b = 1, then dlen ≥ 9.

af, work

Arrays of size laf and (lwork), respectively. The array af contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function p?pttrf and is stored in af.

The array work is a workspace array.

laf

(local) The size of the array af.

Must be laf≥nb_a+2.

If laf is not large enough, an error code is returned and the minimum acceptable size will be returned in af[0].

lwork

(local or global) The size of the array work, must be at least

lwork ≥ (10+2*min(100,nrhs))*NPCOL+4*nrhs.

Output Parameters

b

On exit, this array contains the local pieces of the solution distributed matrix X.

work[0])

On exit, work[0] contains the minimum value of lwork required for optimum performance.

info

If info=0, the execution is successful.

info < 0:

if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.
p?trtrs
Solves a system of linear equations with a triangular distributed matrix.

Syntax

void pstrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pdtrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pctrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pztrtrs (char *uplo , char *trans , char *diag , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

Include Files

• mkl_scalapack.h

Description

The p?trtrsfunction solves for X one of the following systems of linear equations:

sub(A)*X = sub(B),
(sub(A))T*X = sub(B), or
(sub(A))H*X = sub(B),

where sub(A) = A(ia:ia+n-1, ja:ja+n-1) is a triangular distributed matrix of order n, and sub(B) denotes the distributed matrix B(ib:ib+n-1, jb:jb+nrhs-1).

A check is made to verify that sub(A) is nonsingular.

Input Parameters

uplo  
(global) Must be 'U' or 'L'.
Indicates whether sub(A) is upper or lower triangular:
If uplo = 'U', then sub(A) is upper triangular.
If uplo = 'L', then sub(A) is lower triangular.

trans  
(global) Must be 'N' or 'T' or 'C'.
Indicates the form of the equations:
If trans = 'N', then sub(A)*X = sub(B) is solved for X.
If trans = 'T', then sub(A)T*X = sub(B) is solved for X.
If trans = 'C', then sub(A)H*X = sub(B) is solved for X.

diag  
(global) Must be 'N' or 'U'.
If diag = 'N', then sub(A) is not a unit triangular matrix.
If \( \text{diag} = 'U' \), then \( \text{sub}(A) \) is unit triangular.

\( n \)  
(global) The order of the distributed matrix \( \text{sub}(A) \) \((n \geq 0)\).

\( \text{nrhs} \)  
(global) The number of right-hand sides; i.e., the number of columns of the distributed matrix \( \text{sub}(B) \) \((\text{nrhs} \geq 0)\).

\( a, b \)  
(local)  
Pointers into the local memory to arrays of local sizes \( \text{lld}_a * \text{LOCc}(ja+n-1) \) and \( \text{lld}_b * \text{LOCc}(jb+\text{nrhs}-1) \), respectively.

The array \( a \) contains the local pieces of the distributed triangular matrix \( \text{sub}(A) \).

If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular matrix, and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced.

If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( \text{sub}(A) \) contains the lower triangular matrix, and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

If \( \text{diag} = 'U' \), the diagonal elements of \( \text{sub}(A) \) are also not referenced and are assumed to be 1.

On entry, the array \( b \) contains the local pieces of the right hand side distributed matrix \( \text{sub}(B) \).

\( ia, ja \)  
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( \text{desca} \)  
(global and local) array of size \( \text{dlen}_a \). The array descriptor for the distributed matrix \( A \).

\( ib, jb \)  
(global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the matrix \( \text{sub}(B) \), respectively.

\( \text{descb} \)  
(global and local) array of size \( \text{dlen}_b \). The array descriptor for the distributed matrix \( B \).

**Output Parameters**

\( b \)  
On exit, if \( \text{info}=0 \), \( \text{sub}(B) \) is overwritten by the solution matrix \( X \).

\( \text{info} \)  
If \( \text{info}=0 \), the execution is successful.

\( \text{info} < 0 \):  
if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

\( \text{info} > 0 \):  
if \( \text{info} = i \), the \( i \)-th diagonal element of \( \text{sub}(A) \) is zero, indicating that the submatrix is singular and the solutions \( X \) have not been computed.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.
Estimating the Condition Number: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for estimating the condition number of a matrix. The condition number is used for analyzing the errors in the solution of a system of linear equations. Since the condition number may be arbitrarily large when the matrix is nearly singular, the routines actually compute the reciprocal condition number.

**p?gecon**

Estimates the reciprocal of the condition number of a general distributed matrix in either the 1-norm or the infinity-norm.

**Syntax**

```c
void psgecon (char *norm, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *anorm, float *rcond, float *work, MKL_INT *iwork, MKL_INT *iwork, MKL_INT *info);

void pdgecon (char *norm, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *anorm, double *rcond, double *work, MKL_INT *iwork, MKL_INT *iwork, MKL_INT *info);

void pcgecon (char *norm, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *anorm, float *rcond, MKL_Complex8 *work, MKL_INT *iwork, float *rwork, MKL_INT *iwork, MKL_INT *info);

void pzgecon (char *norm, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *anorm, double *rcond, MKL_Complex16 *work, MKL_INT *iwork, double *rwork, MKL_INT *iwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?gecon function estimates the reciprocal of the condition number of a general distributed real/complex matrix \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) in either the 1-norm or infinity-norm, using the LU factorization computed by p?getrf.

An estimate is obtained for \( \|\text{sub}(A)^{-1}\| \), and the reciprocal of the condition number is computed as

\[
\text{rcond} = \frac{1}{\|\text{sub}(A)\| \times \|\text{sub}(A)^{-1}\|}
\]

**Input Parameters**

- **norm**
  (global) Must be '1' or 'O' or 'I'.
  Specifies whether the 1-norm condition number or the infinity-norm condition number is required.
  If \( \text{norm} = '1' \) or '0', then the 1-norm is used;
  If \( \text{norm} = 'I' \), then the infinity-norm is used.

- **n**
  (global) The order of the distributed matrix \( \text{sub}(A) \) \((n \geq 0)\).
a
 (local)
 Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1).

The array a contains the local pieces of the factors L and U from the
factorization sub(A) = P*L*U; the unit diagonal elements of L are not
stored.

ia, ja
 (global) The row and column indices in the global matrix A indicating the
first row and the first column of the matrix sub(A), respectively.

desca
 (global and local) array of size dlen_. The array descriptor for the
distributed matrix A.

anorm
 (global)
 If norm = '1' or 'O', the 1-norm of the original distributed matrix sub(A);
If norm = 'I', the infinity-norm of the original distributed matrix sub(A).

work
 (local)
 The array work of size lwork is a workspace array.

lwork
 (local or global) The size of the array work.
 For real flavors:
 lwork must be at least
 lwork\geq 2*LOCr(n+mod(ia-1,mb_a))+2*LOCc(n+mod(ja-1,nb_a))
 +\max(2, \max(nb_a*\max(1,iceil(NPROW-1, NPCOL))), LOCc(n
 +mod(ja-1,nb_a)) + nb_a*\max(1,iceil(NPCOL-1, NPROW))).

 For complex flavors:
 lwork must be at least
 lwork\geq 2*LOCr(n+mod(ia-1,mb_a))+\max(2,
 max(nb_a*iceil(NPROW-1, NPCOL), LOCc(n+mod(ja-1,nb_a)) +
 nb_a*iceil(NPCOL-1, NPROW))).

LOCr and LOCc values can be computed using the ScaLAPACK tool function
numroc; NPROW and NPCOL can be determined by calling the function
blacs_gridinfo.

NOTE
 iceil(x,y) is the ceiling of x/y, and mod(x,y) is the integer
 remainder remainder of x/y.

iwork
 (local) Workspace array of size liwork. Used in real flavors only.

liwork
 (local or global) The size of the array iwork; used in real flavors only. Must
be at least
 liwork\geq LOCr(n+mod(ia-1,mb_a)).

rwork
 (local)
 Workspace array of size lrwork. Used in complex flavors only.
lrwork  
(local or global) The size of the array rwork; used in complex flavors only. 
Must be at least 
\[ lrwork \geq \max(1, 2*\text{LOCc}(n+\text{mod}(ja-1, nb_a))). \]

**Output Parameters**

rcond  
(global) 
The reciprocal of the condition number of the distributed matrix sub(A). See 
Description.

work[0]  
On exit, work[0] contains the minimum value of lwork required for 
optimum performance.

iwork[0]  
On exit, iwork[0] contains the minimum value of liwork required for 
optimum performance (for real flavors).

rwork[0]  
On exit, rwork[0] contains the minimum value of lrwork required for 
optimum performance (for complex flavors).

info  
(global) If info=0, the execution is successful. 
info < 0: 
If the i-th argument is an array and the j-th entry, indexed j - 1, had an 
illegal value, then info = -(i*100+j); if the i-th argument is a scalar and 
had an illegal value, then info = -i.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?pocon**

Estimates the reciprocal of the condition number (in 
the 1 - norm) of a symmetric / Hermitian positive-
definite distributed matrix.

**Syntax**

```c
void pspocon (char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *
desca, float *anorm, float *rcond, float *work, MKL_INT *lwork, MKL_INT *iwork,
MKL_INT *info);
void pdpocon (char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *
desca, double *anorm, double *rcond, double *work, MKL_INT *lwork, MKL_INT *
iwork, MKL_INT *iwork, MKL_INT *info);
void pcpocon (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja,
MKL_INT *desca, float *anorm, float *rcond, MKL_Complex8 *work, MKL_INT *
lwork, float *rwork, MKL_INT *lrwork, MKL_INT *info);
void pzpocon (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja,
MKL_INT *desca, double *anorm, double *rcond, MKL_Complex16 *work, MKL_INT *
lwork, double *rwork, MKL_INT *lrwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h
Description

The `p?pocon` function estimates the reciprocal of the condition number (in the 1 - norm) of a real symmetric or complex Hermitian positive definite distributed matrix `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`, using the Cholesky factorization `sub(A) = U^H*U` or `sub(A) = L*L^H` computed by `p?potrf`.

An estimate is obtained for `||sub(A)^{-1}||`, and the reciprocal of the condition number is computed as

$$rcond = \frac{1}{\|\text{Sub}(A)\| \times \|\text{Sub}(A)^{-1}\|}$$

Input Parameters

**uplo**

(global) Must be `'U'` or `'L'`.

Specifies whether the factor stored in `sub(A)` is upper or lower triangular.

If `uplo = 'U'`, `sub(A)` stores the upper triangular factor `U` of the Cholesky factorization `sub(A) = U^H*U`.

If `uplo = 'L'`, `sub(A)` stores the lower triangular factor `L` of the Cholesky factorization `sub(A) = L*L^H`.

**n**

(global) The order of the distributed matrix `sub(A)` (n\geq0).

**a**

(local)

Pointer into the local memory to an array of size `lld_a*LOCc(ja+n-1)`.

The array `a` contains the local pieces of the factors `L` or `U` from the Cholesky factorization `sub(A) = U^H*U`, or `sub(A) = L*L^H`, as computed by `p?potrf`.

**ia, ja**

(global) The row and column indices in the global matrix `A` indicating the first row and the first column of the matrix `sub(A)`, respectively.

**desca**

(global and local) array of size `dlen_`. The array descriptor for the distributed matrix `A`.

**anorm**

(global)

The 1-norm of the symmetric/Hermitian distributed matrix `sub(A)`.

**work**

(local)

The array `work` of size `lwork` is a workspace array.

**lwork**

(local or global) The size of the array `work`.

*For real flavors:*

`lwork` must be at least

$$lwork \geq 2*\text{LOCr}(n+\text{mod}(ia-1,mb_a))+2*\text{LOCc}(n+\text{mod}(ja-1,nb_a)) + \text{max}(2, \text{max}(nb_a*\text{iceil}(\text{NPROW}-1, \text{NPCOL}), \text{LOCc}(n+\text{mod}(ja-1,nb_a))+nb_a*\text{iceil}(\text{NPCOL}-1, \text{NPROW})))$$

*For complex flavors:*

`lwork` must be at least

$$lwork \geq 2*\text{LOCr}(n+\text{mod}(ia-1,mb_a))+2*\text{LOCr}(n+\text{mod}(ja-1,nb_a)) + \text{max}(2, \text{max}(nb_a*\text{iceil}(\text{NPROW}-1, \text{NPCOL}), \text{LOCr}(n+\text{mod}(ja-1,nb_a))+nb_a*\text{iceil}(\text{NPCOL}-1, \text{NPROW})))$$
\[ lwork \geq 2 \cdot \text{LOCr}(n + \text{mod}(ia - 1, mb_a)) + \text{max}(2, \text{max}(nb_a \cdot \text{max}(1, \text{ceil}(\text{NPROW} - 1, \text{NPCOL})), \text{LOCc}(n + \text{mod}(ja - 1, nb_a)) + nb_a \cdot \text{max}(1, \text{ceil}(\text{NPCOL} - 1, \text{NPROW})))) \].

If \( lwork = -1 \), then \( lwork \) is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by \text{p}x\text{erbla}.

**NOTE**

\( \text{ceil}(x, y) \) is the ceiling of \( x/y \), and \( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

\( iwork \) (local) Workspace array of size \( liwork \). Used in real flavors only.

\( liwork \) (local or global) The size of the array \( lwork \); used in real flavors only. Must be at least \( liwork \geq \text{LOCr}(n + \text{mod}(ia - 1, mb_a)) \).

If \( liwork = -1 \), then \( liwork \) is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by \text{p}x\text{erbla}.

\( rwork \) (local)

Workspace array of size \( lrwork \). Used in complex flavors only.

\( lrwork \) (local or global) The size of the array \( rwork \); used in complex flavors only. Must be at least \( lrwork \geq 2 \cdot \text{LOCc}(n + \text{mod}(ja - 1, nb_a)) \).

If \( lrwork = -1 \), then \( lrwork \) is a global input and a workspace query is assumed. The routine only calculates the minimum and optimal size for all work arrays. Each value is returned in the first entry of the corresponding work array, and no error message is issued by \text{p}x\text{erbla}.

**Output Parameters**

\( rcond \) (global)

The reciprocal of the condition number of the distributed matrix sub(\( A \)).

\( \text{work}[0] \) On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{iwork}[0] \) On exit, \( \text{iwork}[0] \) contains the minimum value of \( liwork \) required for optimum performance (for real flavors).

\( \text{rwork}[0] \) On exit, \( \text{rwork}[0] \) contains the minimum value of \( lrwork \) required for optimum performance (for complex flavors).

\( info \) (global) If \( info = 0 \), the execution is successful.

\( info < 0: \)

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i\times100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?trcon}

Estimates the reciprocal of the condition number of a triangular distributed matrix in either 1-norm or infinity-norm.

**Syntax**

```c
void pstrcon(char *norm, char *uplo, char *diag, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *rcond, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *info);

void pdtrcon(char *norm, char *uplo, char *diag, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *rcond, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *info);

void pctrcon(char *norm, char *uplo, char *diag, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *rcond, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *iwork, MKL_INT *info);

void pztrcon(char *norm, char *uplo, char *diag, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *rcond, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *iwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The \textbf{p?trcon} function estimates the reciprocal of the condition number of a triangular distributed matrix

\[ \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1), \]

in either the 1-norm or the infinity-norm.

The norm of \text{sub}(A) is computed and an estimate is obtained for \( \| \text{sub}(A)^{-1} \| \), then the reciprocal of the condition number is computed as

\[ rcond = \frac{1}{\| \text{sub}(A) \| \times \| \text{sub}(A)^{-1} \|} \]

**Input Parameters**

\textbf{norm} (global) Must be 'I' or 'O' or 'I'.

Specifies whether the 1-norm condition number or the infinity-norm condition number is required.

If \text{norm} = '1' or 'O', then the 1-norm is used;

If \text{norm} = 'I', then the infinity-norm is used.

\textbf{uplo} (global) Must be 'U' or 'L'.

If \text{uplo} = 'U', \text{sub}(A) is upper triangular. If \text{uplo} = 'L', \text{sub}(A) is lower triangular.

\textbf{diag} (global) Must be 'N' or 'U'.
If $\text{diag} = 'N'$, sub($A$) is non-unit triangular. If $\text{diag} = 'U'$, sub($A$) is unit triangular.

$n$ (global) The order of the distributed matrix sub($A$), ($n \geq 0$).

$a$ (local) Pointer into the local memory to an array of size $\text{lld}_a \ast \text{LOCc}(ja+n-1)$.

The array $a$ contains the local pieces of the triangular distributed matrix sub($A$).

If $\text{uplo} = 'U'$, the leading $n$-by-$n$ upper triangular part of this distributed matrix contains the upper triangular matrix, and its strictly lower triangular part is not referenced.

If $\text{uplo} = 'L'$, the leading $n$-by-$n$ lower triangular part of this distributed matrix contains the lower triangular matrix, and its strictly upper triangular part is not referenced.

If $\text{diag} = 'U'$, the diagonal elements of sub($A$) are also not referenced and are assumed to be 1.

$ia, ja$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the matrix sub($A$), respectively.

$\text{desca}$ (global and local) array of size $\text{dlen}_\_$. The array descriptor for the distributed matrix $A$.

$\text{work}$ (local) The array $\text{work}$ of size $\text{lwork}$ is a workspace array.

$lwork$ (local or global) The size of the array $\text{work}$.

For real flavors:

$lwork$ must be at least

$$lwork \geq 2 \ast \text{LOCr}(n+\text{mod}(ia-1,mb_a))+\text{LOCc}(n+\text{mod}(ja-1,nb_a))+\max(2,\max(\text{nb}_a\ast\max(1,\text{iceil}(\text{NPROW}-1,\text{NPCOL})),\text{LOCc}(n+\text{mod}(ja-1,nb_a))+\text{nb}_a\ast\max(1,\text{iceil}(\text{NPCOL}-1,\text{NPROW})))).$$

For complex flavors:

$lwork$ must be at least

$$lwork \geq 2 \ast \text{LOCr}(n+\text{mod}(ia-1,mb_a))+\max(2,\max(\text{nb}_a\ast\text{iceil}(\text{NPROW}-1,\text{NPCOL}),\text{LOCc}(n+\text{mod}(ja-1,nb_a))+\text{nb}_a\ast\text{iceil}(\text{NPCOL}-1,\text{NPROW}))).$$

**NOTE**

$\text{iceil}(x,y)$ is the ceiling of $x/y$, and $\text{mod}(x,y)$ is the integer remainder of $x/y$.

$iwork$ (local) Workspace array of size $\text{liwork}$. Used in real flavors only.

$\text{liwork}$ (local or global) The size of the array $\text{iwork}$; used in real flavors only. Must be at least
liwork $\geq$ LOCr($n$+$\text{mod}(ia-1,mb_a)$).

**rwork**

(local)
Workspace array of size lrwork. Used in complex flavors only.

**lrwork**

(local or global) The size of the array rwork; used in complex flavors only. Must be at least

lrwork $\geq$ LOCc($n$+$\text{mod}(ja-1,nb_a)$).

**Output Parameters**

**rcond**

(global)
The reciprocal of the condition number of the distributed matrix sub($A$).

**work[0]**

On exit, work[0] contains the minimum value of lwork required for optimum performance.

**iwork[0]**

On exit, iwork[0] contains the minimum value of liwork required for optimum performance (for real flavors).

**rwork[0]**

On exit, rwork[0] contains the minimum value of lrwork required for optimum performance (for complex flavors).

**info**

(global) If info=0, the execution is successful.

info $<$ 0:
If the $i$-th argument is an array and the $j$-th entry, indexed $j$ - 1, had an illegal value, then info = -(i$\times$100$+$j); if the $i$-th argument is a scalar and had an illegal value, then info = -$i$.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

### Refining the Solution and Estimating Its Error: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for refining the computed solution of a system of linear equations and estimating the solution error. You can call these routines after factorizing the matrix of the system of equations and computing the solution (see Routines for Matrix Factorization and Solving Systems of Linear Equations).

**p?gerfs**

Improves the computed solution to a system of linear equations and provides error bounds and backward error estimates for the solution.

**Syntax**

void p?gerfs (char *trans , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descf , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
Include Files

- mkl_scalapack.h

Description

The `pdgerfs` function improves the computed solution to one of the systems of linear equations

\[
\text{sub}(A) \cdot \text{sub}(X) = \text{sub}(B),
\]

\[
\text{sub}(A)^T \cdot \text{sub}(X) = \text{sub}(B), \text{ or }
\]

\[
\text{sub}(A)^H \cdot \text{sub}(X) = \text{sub}(B)
\]

and provides error bounds and backward error estimates for the solution.

Here \( \text{sub}(A) = A(\text{ia:ia}+n-1, \text{ja:ja}+n-1) \), \( \text{sub}(B) = B(\text{ib:ib}+n-1, \text{jb:jb}+\text{nrhs}-1) \), and \( \text{sub}(X) = X(\text{ix:ix}+n-1, \text{ix:xj}+\text{nrhs}-1) \).

Input Parameters

- `trans` (global) Must be 'N' or 'T' or 'C'.
  
  Specifies the form of the system of equations:
  
  If `trans = 'N'`, the system has the form \( \text{sub}(A) \cdot \text{sub}(X) = \text{sub}(B) \) (No transpose);
  
  If `trans = 'T'`, the system has the form \( \text{sub}(A)^T \cdot \text{sub}(X) = \text{sub}(B) \) (Transpose);
  
  If `trans = 'C'`, the system has the form \( \text{sub}(A)^H \cdot \text{sub}(X) = \text{sub}(B) \) (Conjugate transpose).

- `n` (global) The order of the distributed matrix \( \text{sub}(A) \) (\( n \geq 0 \)).

- `nrhs` (global) The number of right-hand sides, i.e., the number of columns of the matrices \( \text{sub}(B) \) and \( \text{sub}(X) \) (\( \text{nrhs} \geq 0 \)).

- `a, af, b, x` (local)

  Pointers into the local memory to arrays of local sizes

  \( a: \text{ld}_a \cdot \text{LOCc}(\text{ja}+n-1), \)
\[ af: \text{lld}_af * \text{LOCc}(jaf+n-1), \]
\[ b: \text{lld}_b * \text{LOCc}(jb+nrhs-1), \]
\[ x: \text{lld}_x * \text{LOCc}(jx+nrhs-1). \]

The array \( a \) contains the local pieces of the distributed matrix \( \text{sub}(A) \).

The array \( af \) contains the local pieces of the distributed factors of the matrix \( \text{sub}(A) = P*L*U \) as computed by \( \text{pgetrf} \).

The array \( b \) contains the local pieces of the distributed matrix of right hand sides \( \text{sub}(B) \).

On entry, the array \( x \) contains the local pieces of the distributed solution matrix \( \text{sub}(X) \).

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( desca \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( A \).

\( iaf, jaf \) (global) The row and column indices in the global matrix \( AF \) indicating the first row and the first column of the matrix \( \text{sub}(AF) \), respectively.

\( descaf \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( AF \).

\( ib, jb \) (global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the matrix \( \text{sub}(B) \), respectively.

\( descb \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( B \).

\( ix, jx \) (global) The row and column indices in the global matrix \( X \) indicating the first row and the first column of the matrix \( \text{sub}(X) \), respectively.

\( descx \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( X \).

\( ipiv \) (local)

Array of size \( \text{LOCr}(m_af) + mb_af \).

This array contains pivoting information as computed by \( \text{pgetrf} \). If \( ipiv[i]=j \), then the local row \( i+1 \) was swapped with the global row \( j \) where \( i=0, \ldots, \text{LOCr}(m_af) + mb_af-1 \).

This array is tied to the distributed matrix \( A \).

\( work \) (local)

The array \( work \) of size \( lwork \) is a workspace array.

\( lwork \) (local or global) The size of the array \( work \).

For real flavors:

\( lwork \) must be at least

\( lwork \geq 3*\text{LOCr}(n+\text{mod}(ia-1,mb_a)) \)

For complex flavors:
`lwork` must be at least
\[ lwork \geq 2 \times \text{LOCr}(n + \text{mod}(ia-1, mb_a)) \]

**NOTE**

\[ \text{mod}(x, y) \] is the integer remainder of \( x/y \).

- `iwork` (local) Workspace array, size `liwork`. Used in real flavors only.
- `liwork` (local or global) The size of the array `lwork`; used in real flavors only. Must be at least
\[ liwork \geq \text{LOCr}(n + \text{mod}(ib-1, mb_b)) \]
- `rwork` (local)
  Workspace array, size `lrwork`. Used in complex flavors only.
- `lrwork` (local or global) The size of the array `rwork`; used in complex flavors only. Must be at least
\[ lrwork \geq \text{LOCr}(n + \text{mod}(ib-1, mb_b)) \]

**Output Parameters**

- `x`
  Arrays of size `LOCc(jb+nrhs-1)` each.
  On exit, contains the improved solution vectors.

- `ferr, berr`
  The array `ferr` contains the estimated forward error bound for each solution vector of sub(\(X\)).
  If XTRUE is the true solution corresponding to sub(\(X\)), `ferr` is an estimated upper bound for the magnitude of the largest element in (sub(\(X\)) - XTRUE) divided by the magnitude of the largest element in sub(\(X\)). The estimate is as reliable as the estimate for `rcond`, and is almost always a slight overestimate of the true error.
  This array is tied to the distributed matrix \(X\).

  The array `berr` contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(\(A\)) or sub(\(B\)) that makes sub(\(X\)) an exact solution). This array is tied to the distributed matrix \(X\).

- `work[0]`
  On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

- `iwork[0]`
  On exit, `iwork[0]` contains the minimum value of `liwork` required for optimum performance (for real flavors).

- `rwork[0]`
  On exit, `rwork[0]` contains the minimum value of `lrwork` required for optimum performance (for complex flavors).

- `info`
  (global) If `info=0`, the execution is successful.

  - `info < 0`:
    If the \(i\)-th argument is an array and the \(j\)-th entry, indexed \( j - 1 \), had an illegal value, then `info = -(i*100+j)`; if the \(i\)-th argument is a scalar and had an illegal value, then `info = -i`.  

1280
**Overview**
for details of ScaLAPACK array descriptor structures and related notations.

**p?porfs**

Improves the computed solution to a system of linear equations with symmetric/Hermitian positive definite distributed matrix and provides error bounds and backward error estimates for the solution.

**Syntax**

```c
void pspors (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descf , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

```c
void pdporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descf , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
```

```c
void pcporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descf , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *ferr , float *berr , MKL_Complex8 *work , MKL_INT *iwork , float *rwork , MKL_INT *lrwork , MKL_INT *info );
```

```c
void pzporfs (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descf , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?porfs function improves the computed solution to the system of linear equations

\[
\text{sub}(A)*\text{sub}(X) = \text{sub}(B),
\]

where \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) is a real symmetric or complex Hermitian positive definite distributed matrix and

\[
\text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1),
\]

\[
\text{sub}(X) = X(ix:ix+n-1, jx:jx+nrhs-1)
\]

are right-hand side and solution submatrices, respectively. This function also provides error bounds and backward error estimates for the solution.

**Input Parameters**

- **uplo**
  (global) Must be 'U' or 'L'.

---

*ScaLAPACK Routines* 4

1281
Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored.

If uplo = 'U', sub(A) is upper triangular. If uplo = 'L', sub(A) is lower triangular.

\( n \) (global) The order of the distributed matrix sub(A) \( (n \geq 0) \).

\( nrhs \) (global) The number of right-hand sides, i.e., the number of columns of the matrices sub(B) and sub(X) \( (nrhs \geq 0) \).

\( a, af, b, x \) (local)

Pointers into the local memory to arrays of local sizes

- \( a: lld_a \times LOCc(ja+n-1) \),
- \( af: lld_{af} \times LOCc(jaf+n-1) \),
- \( b: lld_b \times LOCc(jb+nrhs-1) \),
- \( x: lld_x \times LOCc(jx+nrhs-1) \).

The array \( a \) contains the local pieces of the \( n \)-by-\( n \) symmetric/Hermitian distributed matrix sub(A).

If uplo = 'U', the leading \( n \)-by-\( n \) upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If uplo = 'L', the leading \( n \)-by-\( n \) lower triangular part of sub(A) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

The array \( af \) contains the factors \( L \) or \( U \) from the Cholesky factorization sub(A) = \( L^H \times L \) or sub(A) = \( U \times U^H \), as computed by \( p?potrf \).

On entry, the array \( b \) contains the local pieces of the distributed matrix of right hand sides sub(B).

On entry, the array \( x \) contains the local pieces of the solution vectors sub(X).

\( ia, ja \) (global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

\( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix A.

\( iaf, jaf \) (global) The row and column indices in the global matrix AF indicating the first row and the first column of the matrix sub(AF), respectively.

\( descaf \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix AF.

\( ib, jb \) (global) The row and column indices in the global matrix B indicating the first row and the first column of the matrix sub(B), respectively.

\( descb \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix B.

\( ix, jx \) (global) The row and column indices in the global matrix X indicating the first row and the first column of the matrix sub(X), respectively.
(global and local) array of size dlen_. The array descriptor for the distributed matrix X.

(work (local)
The array work of size lwork is a workspace array.

(lwork (local) The size of the array work.

For real flavors:
lwork must be at least
lwork \geq 3*LOCr(n+\text{mod}(ia-1,mb_a))

For complex flavors:
lwork must be at least
lwork \geq 2*LOCr(n+\text{mod}(ia-1,mb_a))

NOTE
\text{mod}(x,y) is the integer remainder of x/y.

(iwork (local) Workspace array of size liwork. Used in real flavors only.

(liwork (local or global) The size of the array iwork; used in real flavors only. Must be at least
liwork \geq \text{LOCr}(n+\text{mod}(ib-1,mb_b)).

(rwork (local)
Workspace array of size lrwork. Used in complex flavors only.

(lrwork (local or global) The size of the array rwork; used in complex flavors only. Must be at least lrwork \geq \text{LOCr}(n+\text{mod}(ib-1,mb_b)).

Output Parameters

x

On exit, contains the improved solution vectors.

Arrays of size LOCc(jb+nrhs-1) each.

The array ferr contains the estimated forward error bound for each solution vector of sub(X).

If XTRUE is the true solution corresponding to sub(X), ferr is an estimated upper bound for the magnitude of the largest element in (sub(X) - XTRUE) divided by the magnitude of the largest element in sub(X). The estimate is as reliable as the estimate for rcond, and is almost always a slight overestimate of the true error.

This array is tied to the distributed matrix X.

The array berr contains the component-wise relative backward error of each solution vector (that is, the smallest relative change in any entry of sub(A) or sub(B) that makes sub(X) an exact solution). This array is tied to the distributed matrix X.
On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

On exit, `iwork[0]` contains the minimum value of `liwork` required for optimum performance (for real flavors).

On exit, `rwork[0]` contains the minimum value of `lrwork` required for optimum performance (for complex flavors).

(info) If `info=0`, the execution is successful.

If the `i`-th argument is an array and the `j`-th entry, indexed `j-1`, had an illegal value, then `info = -(i*100+j)`; if the `i`-th argument is a scalar and had an illegal value, then `info = -i`.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?trrfs**
Provides error bounds and backward error estimates for the solution to a system of linear equations with a distributed triangular coefficient matrix.

**Syntax**

```c
void pstrrfs (char *uplo, char *trans, char *diag, MKL_INT *n, MKL_INT *nrhs, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *ferr, float *berr, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *info);

void pdtrrfs (char *uplo, char *trans, char *diag, MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *ferr, double *berr, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);

void pctrrfs (char *uplo, char *trans, char *diag, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *ferr, float *berr, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *lrwork, MKL_INT *info);

void pztrrfs (char *uplo, char *trans, char *diag, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *ferr, double *berr, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *lrwork, MKL_INT *info);
```

**Include Files**
- mkl_scalapack.h

**Description**
The `p?trrfs` function provides error bounds and backward error estimates for the solution to one of the systems of linear equations.
\( \text{sub}(A)*\text{sub}(X) = \text{sub}(B), \)
\( \text{sub}(A)^T*\text{sub}(X) = \text{sub}(B), \) or
\( \text{sub}(A)^H*\text{sub}(X) = \text{sub}(B), \)
where \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) is a triangular matrix,
\( \text{sub}(B) = B(ib:ib+n-1, jb:jb+\text{nrhs}-1), \) and
\( \text{sub}(X) = X(ix:ix+n-1, jx:jx+\text{nrhs}-1). \)

The solution matrix \( X \) must be computed by \texttt{p?trtrs} or some other means before entering this function. The function \texttt{p?trrfs} does not do iterative refinement because doing so cannot improve the backward error.

**Input Parameters**

- **uplo**
  
  (global) Must be 'U' or 'L'.
  
  If \( \text{uplo} = 'U' \), \( \text{sub}(A) \) is upper triangular. If \( \text{uplo} = 'L' \), \( \text{sub}(A) \) is lower triangular.

- **trans**
  
  (global) Must be 'N' or 'T' or 'C'.
  
  Specifies the form of the system of equations:
  
  If \( \text{trans} = 'N' \), the system has the form \( \text{sub}(A)*\text{sub}(X) = \text{sub}(B) \) (No transpose);
  
  If \( \text{trans} = 'T' \), the system has the form \( \text{sub}(A)^T*\text{sub}(X) = \text{sub}(B) \) (Transpose);
  
  If \( \text{trans} = 'C' \), the system has the form \( \text{sub}(A)^H*\text{sub}(X) = \text{sub}(B) \) (Conjugate transpose).

- **diag**
  
  Must be 'N' or 'U'.
  
  If \( \text{diag} = 'N' \), then \( \text{sub}(A) \) is non-unit triangular.
  
  If \( \text{diag} = 'U' \), then \( \text{sub}(A) \) is unit triangular.

- **n**
  
  (global) The order of the distributed matrix \( \text{sub}(A) \) \((n\geq0)\).

- **nrhs**
  
  (global) The number of right-hand sides, that is, the number of columns of the matrices \( \text{sub}(B) \) and \( \text{sub}(X) \) \((\text{nrhs}\geq0)\).

- **a, b, x**
  
  (local)
  
  Pointers into the local memory to arrays of local sizes
  
  \( a: \text{lld}_a * \text{LOCc}(ja+n-1), \)
  
  \( b: \text{lld}_b * \text{LOCc}(jb+\text{nrhs}-1), \)
  
  \( x: \text{lld}_x * \text{LOCc}(jx+\text{nrhs}-1). \)

  The array \( a \) contains the local pieces of the original triangular distributed matrix \( \text{sub}(A) \).
  
  If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

  If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( \text{sub}(A) \) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.
If diag = 'U', the diagonal elements of sub(A) are also not referenced and are assumed to be 1.

On entry, the array b contains the local pieces of the distributed matrix of right hand sides sub(B).

On entry, the array x contains the local pieces of the solution vectors sub(X).

ia, ja (global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

ib, jb (global) The row and column indices in the global matrix B indicating the first row and the first column of the matrix sub(B), respectively.

descb (global and local) array of size dlen_. The array descriptor for the distributed matrix B.

ix, jx (global) The row and column indices in the global matrix X indicating the first row and the first column of the matrix sub(X), respectively.

descx (global and local) array of size dlen_. The array descriptor for the distributed matrix X.

work (local) The array work of size lwork is a workspace array.

lwork (local) The size of the array work.

For real flavors:

lwork must be at least \( lwork \geq 3 \cdot \text{LOCr}(n + \text{mod}(ia-1, mb_a)) \)

For complex flavors:

lwork must be at least

\( lwork \geq 2 \cdot \text{LOCr}(n + \text{mod}(ia-1, mb_a)) \)

---

**NOTE**

\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

---

iwork (local) Workspace array of size liwork. Used in real flavors only.

liwork (local or global) The size of the array lwork; used in real flavors only. Must be at least

\( liwork \geq \text{LOCr}(n + \text{mod}(ib-1, mb_b)) \).

rwork (local) Workspace array of size lrwork. Used in complex flavors only.

lrwork (local or global) The size of the array rwork; used in complex flavors only. Must be at least \( lrwork \geq \text{LOCr}(n + \text{mod}(ib-1, mb_b)) \).
Output Parameters

**ferr, berr**  
Arrays of size $\text{LOCc}(jb+nrhs-1)$ each.  
The array *ferr* contains the estimated forward error bound for each  
solution vector of $\text{sub}(X)$.  
If $X_{\text{TRUE}}$ is the true solution corresponding to $\text{sub}(X)$, *ferr* is an estimated  
upper bound for the magnitude of the largest element in $(\text{sub}(X) - X_{\text{TRUE}})$  
divided by the magnitude of the largest element in $\text{sub}(X)$. The estimate is  
as reliable as the estimate for $rcond$, and is almost always a slight  
overestimate of the true error.  
This array is tied to the distributed matrix $X$.  
The array *berr* contains the component-wise relative backward error of  
each solution vector (that is, the smallest relative change in any entry of  
$\text{sub}(A)$ or $\text{sub}(B)$ that makes $\text{sub}(X)$ an exact solution). This array is tied to  
the distributed matrix $X$.

**work[0]**  
On exit, *work[0]* contains the minimum value of *lwork* required for  
optimum performance.

**iwork[0]**  
On exit, *iwork[0]* contains the minimum value of *liwork* required for  
optimum performance (for real flavors).

**rwork[0]**  
On exit, *rwork[0]* contains the minimum value of *lrwork* required for  
optimum performance (for complex flavors).

**info**  
(global) If info=0, the execution is successful.  
info < 0:  
If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an  
illegal value, then info = $-(i*100+j)$; if the $i$-th argument is a scalar and  
had an illegal value, then info = $-i$.

See Also  
Overview  for details of ScaLAPACK array descriptor structures and related notations.

Matrix Inversion: ScaLAPACK Computational Routines

This sections describes ScaLAPACK routines that compute the inverse of a matrix based on the previously  
obtained factorization. Note that it is not recommended to solve a system of equations $Ax = b$ by first  
computing $A^{-1}$ and then forming the matrix-vector product $x = A^{-1}b$. Call a solver routine instead (see  
Solving Systems of Linear Equations); this is more efficient and more accurate.

**p?getri**  
*Computes the inverse of a LU-factored distributed matrix.*

**Syntax**

```c
void psgetri (MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca,  
MKL_INT *ipiv, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork,  
MKL_INT *info );
```
void pdgetri (MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);

void pcgetri (MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *work, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);

void pzgetri (MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The p?getri function computes the inverse of a general distributed matrix sub(A) = A(ia:ia+n-1, ja:ja+n-1) using the LU factorization computed by p?getrf. This method inverts U and then computes the inverse of sub(A) by solving the system

\[ \text{inv}(\text{sub}(A)) \times L = \text{inv}(U) \]

for \( \text{inv}(\text{sub}(A)) \).

Input Parameters

\[ n \]
(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A) \((n \geq 0)\).

\[ a \]
(local)
Pointer into the local memory to an array of local size \( lld_a \times LOCc(ja+n-1) \).

On entry, the array \( a \) contains the local pieces of the \( L \) and \( U \) obtained by the factorization sub(A) = \( P \times L \times U \) computed by p?getrf.

\[ ia, ja \]
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix sub(A), respectively.

\[ desca \]
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\[ work \]
(local)
The array \( work \) of size \( lwork \) is a workspace array.

\[ lwork \]
(local) The size of the array \( work \). \( lwork \) must be at least \( lwork \geq LOCr(n + \text{mod}(ia-1,mb_a)) \times nb_a \).

NOTE
\( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

The array \( work \) is used to keep at most an entire column block of sub(A).

\[ iwork \]
(local) Workspace array used for physically transposing the pivots, size \( liwork \).

\[ liwork \]
(local or global) The size of the array \( iwork \).
The minimal value $liwork$ of is determined by the following code:

```plaintext
if NPROW == NPCOL then
    liwork = LOCc(n_a + mod(ja-1,nb_a))+ nb_a
else
    liwork = LOCc(n_a + mod(ja-1,nb_a)) +
    max(ceil(ceil(LOCr(m_a)/mb_a)/(lcm/NPROW)),nb_a)
end if
```

where $lcm$ is the least common multiple of process rows and columns ($NPROW$ and $NPCOL$).

**Output Parameters**

- $ipiv$ (local)
  - Array of size $LOCr(m_a)+ mb_a$.
  - This array contains the pivoting information.
  - If $ipiv[i]=j$, then the local row $i+1$ was swapped with the global row $j$ where $i=0, \ldots, LOCr(m_a) + mb_a-1$.
  - This array is tied to the distributed matrix $A$.

- $work[0]$
  - On exit, $work[0]$ contains the minimum value of $lwork$ required for optimum performance.

- $iwork[0]$
  - On exit, $iwork[0]$ contains the minimum value of $liwork$ required for optimum performance.

- $info$ (global)
  - If $info=0$, the execution is successful.
  - $info < 0$:
    - If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.
  - $info > 0$:
    - If $info = i$, the matrix element $U(i,i)$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, and division by zero will occur if it is used to solve a system of equations.

**See Also**

- **Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?potri**

*Computes the inverse of a symmetric/Hermitian positive definite distributed matrix.*

**Syntax**

```c
void pspotri (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```
void pdpotri (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info);
void pcpotri (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *info);
void pzpotri (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

The p?potrf function computes the inverse of a real symmetric or complex Hermitian positive definite distributed matrix \( A = sub(A) = A(ia:ia+n-1, ja:ja+n-1) \) using the Cholesky factorization \( sub(A) = U^H \cdot U \) or \( sub(A) = L \cdot L^H \) computed by p?potrf.

Input Parameters

uplo (global) Must be 'U' or 'L'.
 Specifies whether the upper or lower triangular part of the symmetric/ Hermitian matrix \( sub(A) \) is stored.

If \( uplo = 'U' \), upper triangle of \( sub(A) \) is stored. If \( uplo = 'L' \), lower triangle of \( sub(A) \) is stored.

n (global) The number of rows and columns to be operated on, that is, the order of the distributed matrix \( sub(A) \) \( (n \geq 0) \).

a (local)
 Pointer into the local memory to an array of local size \( lld_a \cdot LOCc(ja+n-1) \).
 On entry, the array \( a \) contains the local pieces of the triangular factor \( U \) or \( L \) from the Cholesky factorization \( sub(A) = U^H \cdot U \), or \( sub(A) = L \cdot L^H \), as computed by p?potrf.

ia, ja (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( sub(A) \), respectively.

desca (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

Output Parameters

a On exit, overwritten by the local pieces of the upper or lower triangle of the (symmetric/Hermitian) inverse of \( sub(A) \).

info (global) If \( info=0 \), the execution is successful.

\( info < 0 \):
If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i \cdot 100 + j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( info > 0 \):
If info = i, the element (i, i) of the factor U or L is zero, and the inverse could not be computed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trtri
Computes the inverse of a triangular distributed matrix.

Syntax
void pstrtri (char *uplo , char *diag , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pdtrtri (char *uplo , char *diag , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pctrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
void pztrtri (char *uplo , char *diag , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );

Include Files
• mkl_scalapack.h

Description
The p?trtri function computes the inverse of a real or complex upper or lower triangular distributed matrix
sub(A) = A(ia:ia+n-1, ja:ja+n-1).

Input Parameters
uplo (global) Must be 'U' or 'L'.
   Specifies whether the distributed matrix sub(A) is upper or lower triangular.
   If uplo = 'U', sub(A) is upper triangular.
   If uplo = 'L', sub(A) is lower triangular.

diag
   Must be 'N' or 'U'.
   Specifies whether or not the distributed matrix sub(A) is unit triangular.
   If diag = 'N', then sub(A) is non-unit triangular.
   If diag = 'U', then sub(A) is unit triangular.

n (global) The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A) (n≥0).

a (local)
   Pointer into the local memory to an array of local size ld_a*LOCc(ja+n-1).
   The array a contains the local pieces of the triangular distributed matrix sub(A).
If uplo = 'U', the leading n-by-n upper triangular part of sub(A) contains the upper triangular matrix to be inverted, and the strictly lower triangular part of sub(A) is not referenced.

If uplo = 'L', the leading n-by-n lower triangular part of sub(A) contains the lower triangular matrix, and the strictly upper triangular part of sub(A) is not referenced.

ia, ja
(global) The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

desca
(global and local) array of size dlen_. The array descriptor for the distributed matrix A.

Output Parameters

a
On exit, overwritten by the (triangular) inverse of the original matrix.

info
(global) If info=0, the execution is successful.

info < 0:
If the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

info > 0:
If info = k, the matrix element A(ia+k-1, ja+k-1) is exactly zero. The triangular matrix sub(A) is singular and its inverse cannot be computed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

Matrix Equilibration: ScaLAPACK Computational Routines

ScaLAPACK routines described in this section are used to compute scaling factors needed to equilibrate a matrix. Note that these routines do not actually scale the matrices.

p?geequ
Computes row and column scaling factors intended to equilibrate a general rectangular distributed matrix and reduce its condition number.

Syntax

void psgeequ (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *r, float *c, float *rowcnd, float *colcnd, float *amax, MKL_INT *info);
void pdgeequ (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *r, double *c, double *rowcnd, double *colcnd, double *amax, MKL_INT *info);
void pcgeequ (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *r, float *c, float *rowcnd, float *colcnd, float *amax, MKL_INT *info);
void pzgeequ (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *r, double *c, double *rowcnd, double *colcnd, double *amax, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?geequ function computes row and column scalings intended to equilibrate an \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) and reduce its condition number. The output array \( r \) returns the row scale factors \( r_i \), and the array \( c \) returns the column scale factors \( c_j \). These factors are chosen to try to make the largest element in each row and column of the matrix \( B \) with elements \( b_{ij} = r_i a_{ij} c_j \) have absolute value 1.

\( r_i \) and \( c_j \) are restricted to be between \( SMLNUM = \text{smallest safe number} \) and \( BIGNUM = \text{largest safe number} \). Use of these scaling factors is not guaranteed to reduce the condition number of \( \text{sub}(A) \) but works well in practice.

\( SMLNUM \) and \( BIGNUM \) are parameters representing machine precision. You can use the ?lamch routines to compute them. For example, compute single precision values of \( SMLNUM \) and \( BIGNUM \) as follows:

\[
\begin{align*}
SMLNUM &= \text{slamch ('s')} \\
BIGNUM &= 1 / SMLNUM
\end{align*}
\]

The auxiliary function p?laqge uses scaling factors computed by p?geequ to scale a general rectangular matrix.

Input Parameters

- \( m \) (global) The number of rows to be operated on, that is, the number of rows of the distributed matrix \( \text{sub}(A) \) (\( m \geq 0 \)).
- \( n \) (global) The number of columns to be operated on, that is, the number of columns of the distributed matrix \( \text{sub}(A) \) (\( n \geq 0 \)).
- \( a \) (local)
  Pointer into the local memory to an array of local size \( lld_a \times \text{LOCc}(ja+n-1) \).
  The array \( a \) contains the local pieces of the \( m \)-by-\( n \) distributed matrix whose equilibration factors are to be computed.
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.
- \( \text{desca} \) (global and local) array of size \( dlen_\_ \). The array descriptor for the distributed matrix \( A \).

Output Parameters

- \( r, c \) (local)
  Arrays of sizes \( \text{LOCr}(m_a) \) and \( \text{LOCc}(n_a) \), respectively.
  If \( info = 0 \), or \( info > ia+m-1 \), \( r[i] \) contain the row scale factors for \( \text{sub}(A) \) for \( ia-1 \leq i < ia+m-1 \). \( r \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( r \) is tied to the distributed matrix \( A \).
If $info = 0$, $c[i]$ contain the column scale factors for $sub(A)$ for $ja-1 \leq i < ja+n-1$. $c$ is aligned with the distributed matrix $A$, and replicated down every process row. $c$ is tied to the distributed matrix $A$.

$\text{rowcnd, colcnd}$ (global)

If $info = 0$ or $info > ia+m-1$, $\text{rowcnd}$ contains the ratio of the smallest $r_i$ to the largest $r_i$ ($ia \leq i \leq ia+m-1$). If $\text{rowcnd} \geq 0.1$ and $amax$ is neither too large nor too small, it is not worth scaling by $r_i$.

If $info = 0$, $\text{colcnd}$ contains the ratio of the smallest $c_j$ to the largest $c_j$ ($ja \leq j \leq ja+n-1$).

If $\text{colcnd} \geq 0.1$, it is not worth scaling by $c_j$.

$\text{amax}$ (global)

Absolute value of the largest matrix element. If $amax$ is very close to overflow or very close to underflow, the matrix should be scaled.

$\text{info}$ (global) If $info = 0$, the execution is successful.

$\text{info} < 0$:

If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

$\text{info} > 0$:

If $info = i$ and

$I \leq m$, the $i$-th row of the distributed matrix $sub(A)$ is exactly zero;

$I > m$, the $(i - m)$-th column of the distributed matrix $sub(A)$ is exactly zero.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p？poequ**

Computes row and column scaling factors intended to equilibrate a symmetric (Hermitian) positive definite distributed matrix and reduce its condition number.

**Syntax**

void pspoequ (MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *sr, float *sc, float *scond, float *amax, MKL_INT *info);

void pdpoequ (MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *sr, double *sc, double *scond, double *amax, MKL_INT *info);

void pcpoequ (MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *sr, float *sc, float *scond, float *amax, MKL_INT *info);

void pzpoequ (MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *sr, double *sc, double *scond, double *amax, MKL_INT *info);
Include Files

- mkl_scalapack.h

Description

The \( \texttt{p?poequ} \) function computes row and column scalings intended to equilibrate a real symmetric or complex Hermitian positive definite distributed matrix \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) and reduce its condition number (with respect to the two-norm). The output arrays \( \text{sr} \) and \( \text{sc} \) return the row and column scale factors:

\[
S(i) = \frac{1}{\sqrt{a_{i,i}}}
\]

These factors are chosen so that the scaled distributed matrix \( B \) with elements \( b_{ij} = s(i) * a_{ij} * s(j) \) has ones on the diagonal. This choice of \( \text{sr} \) and \( \text{sc} \) puts the condition number of \( B \) within a factor \( n \) of the smallest possible condition number over all possible diagonal scalings.

The auxiliary function \( \texttt{p?laqsy} \) uses scaling factors computed by \( \texttt{p?geequ} \) to scale a general rectangular matrix.

Input Parameters

- \( n \) (global) The number of rows and columns to be operated on, that is, the order of the distributed matrix \( \text{sub}(A) \) \((n \geq 0)\).
- \( a \) (local) Pointer into the local memory to an array of local size \( \text{lld}_{a} \ast \text{LOCc}(ja+n-1) \).
  The array \( a \) contains the \( n \)-by-\( n \) symmetric/Hermitian positive definite distributed matrix \( \text{sub}(A) \) whose scaling factors are to be computed. Only the diagonal elements of \( \text{sub}(A) \) are referenced.
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.
- \( \text{desca} \) (global and local) array of size \( \text{dlen}_\_ \). The array descriptor for the distributed matrix \( A \).

Output Parameters

- \( \text{sr}, \text{sc} \) (local) Arrays of sizes \( \text{LOCr}(m_\_) \) and \( \text{LOCc}(n_\_) \), respectively.
  If \( \text{info} = 0 \), the array \( \text{sr}(ia:ia+n-1) \) contains the row scale factors for \( \text{sub}(A) \). \( \text{sr} \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( \text{sr} \) is tied to the distributed matrix \( A \).
  If \( \text{info} = 0 \), the array \( \text{sc}(ja:ja+n-1) \) contains the column scale factors for \( \text{sub}(A) \). \( \text{sc} \) is aligned with the distributed matrix \( A \), and replicated down every process row. \( \text{sc} \) is tied to the distributed matrix \( A \).
- \( \text{scond} \) (global)
If \( info = 0 \), \( scond \) contains the ratio of the smallest \( sr[i] \) (or \( sc[j] \)) to the largest \( sr[i] \) (or \( sc[j] \)), with
\[
1 \leq i < ia+n-1 \quad \text{and} \quad 1 \leq j < ja+n-1.
\]

If \( scond \geq 0.1 \) and \( amax \) is neither too large nor too small, it is not worth scaling by \( sr \) (or \( sc \)).

\( amax \)

Absolute value of the largest matrix element. If \( amax \) is very close to overflow or very close to underflow, the matrix should be scaled.

\( info \)

If \( info = 0 \), the execution is successful.

\( info < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( info > 0 \):

If \( info = k \), the \( k \)-th diagonal entry of \( sub(A) \) is nonpositive.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

Orthogonal Factorizations: ScaLAPACK Computational Routines

This section describes the ScaLAPACK routines for the QR(RQ) and LQ(QL) factorization of matrices. Routines for the RZ factorization as well as for generalized QR and RQ factorizations are also included. For the mathematical definition of the factorizations, see the respective LAPACK sections or refer to [SLUG].

Table "Computational Routines for Orthogonal Factorizations" lists ScaLAPACK routines that perform orthogonal factorization of matrices.

Computational Routines for Orthogonal Factorizations

<table>
<thead>
<tr>
<th>Matrix type, factorization</th>
<th>Factorize without pivoting</th>
<th>Factorize with pivoting</th>
<th>Generate matrix Q</th>
<th>Apply matrix Q</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>p?ungqr</td>
<td>p?unmqr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p?ungqr</td>
<td>p?unmqr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p?ungql</td>
<td>p?unmql</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>p?ungql</td>
<td>p?unmql</td>
</tr>
<tr>
<td>trapezoidal matrices, RZ factorization</td>
<td>p?tzrzf</td>
<td></td>
<td></td>
<td>p?ormrz</td>
</tr>
<tr>
<td>Matrix type, factorization</td>
<td>Factorize without pivoting</td>
<td>Factorize with pivoting</td>
<td>Generate matrix Q</td>
<td>Apply matrix Q</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------</td>
<td>-------------------------</td>
<td>--------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>pair of matrices, generalized QR factorization</td>
<td>( p?ggqrf )</td>
<td>( p?ggqrf )</td>
<td>( p?unmrz )</td>
<td></td>
</tr>
<tr>
<td>pair of matrices, generalized RQ factorization</td>
<td>( p?ggqrf )</td>
<td>( p?ggqrf )</td>
<td>( p?unmrz )</td>
<td></td>
</tr>
</tbody>
</table>

**p?geqrf**

*Computes the QR factorization of a general m-by-n matrix.*

**Syntax**

```c
void pgeqrf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void dgeqrf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void cgeqrf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void zgeqrf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?geqrf` function forms the QR factorization of a general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) as \( A=Q*R \).

**Input Parameters**

- \( m \) (global) The number of rows in the distributed matrix \( \text{sub}(A) \); \( (m \geq 0) \).
- \( n \) (global) The number of columns in the distributed matrix \( \text{sub}(A) \); \( (n \geq 0) \).
- \( \text{a} \) (local) Pointer into the local memory to an array of local size \( lld_a*\text{LOCc}(ja+n-1) \).
- \( \text{ia, ja} \) Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.
- (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.
(global and local) array of size dlen_. The array descriptor for the distributed matrix A

(local).

Workspace array of size lwork.

(local or global) size of work, must be at least lwork≥nb_a * (mp0+mq0+nb_a), where

irow = indxg2p(ia, mb_a, MYROW, rsrca, NPROW),
irow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
irow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
mp0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
ir0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL), and numroc,
indxg2p are ScalAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

The elements on and above the diagonal of sub(A) contain the min(m,n)-by-n upper trapezoidal matrix R (R is upper triangular if m≥n); the elements below the diagonal, with the array tau, represent the orthogonal/unitary matrix Q as a product of elementary reflectors (see Application Notes below).

Array of size LOCc(ja+min(m,n)-1).

Contains the scalar factor of elementary reflectors. tau is tied to the distributed matrix A.

On exit, work[0] contains the minimum value of lwork required for optimum performance.

= 0, the execution is successful.

< 0, if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

Application Notes

The matrix Q is represented as a product of elementary reflectors

Q = H(ja)*H(ja+1)*...*H(ja+k-1),

where k = min(m,n).

Each H(i) has the form
\[ H(i) = I - \tau v^* v \]

where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(1:i-1) = 0 \) and \( v(i) = 1; v(i+1:m) \) is stored on exit in \( A(ia+i:ia+m-1, ja+i-1) \), and \( \tau \) in \( \tau[ja+i-2] \).

**See Also**
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?geqpf**
Computes the QR factorization of a general \( m \)-by-\( n \) matrix with pivoting.

**Syntax**

void psgeqpf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgeqpf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgeqpf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *lrwork, MKL_INT *info);
void pzgeqpf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *lrwork, MKL_INT *info);

**Include Files**

- mkl_scalapack.h

**Description**
The p?geqpf function forms the QR factorization with column pivoting of a general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) as \( \text{sub}(A)*P = Q*R \).

**Input Parameters**

- \( m \) (global) The number of rows in the matrix \( \text{sub}(A) \) \( (m \geq 0) \).
- \( n \) (global) The number of columns in the matrix \( \text{sub}(A) \) \( (n \geq 0) \).
- \( a \) (local)
  Pointer into the local memory to an array of local size \( lld_a*\text{LOCc}(ja+n-1) \). Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.
- \( \text{desca} \) (global and local) array of size \( dlen_. \) The array descriptor for the distributed matrix \( A \).
- \( \text{work} \) (local)
  Workspace array of size \( \text{lwork} \).
(local or global) size of work, must be at least

For real flavors:
\[ lwork \geq \max(3, mp_0 + nq_0) + \text{LOCc } (ja+n-1) + nq_0. \]

For complex flavors:
\[ lwork \geq \max(3, mp_0 + nq_0). \]

Here
\[ iroff = \text{mod}(ia-1, mb_a), \quad icoff = \text{mod}(ja-1, nb_a), \]
\[ iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}), \]
\[ iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}), \]
\[ mp_0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}), \]
\[ nq_0 = \text{numroc}(n+icooff, nb_a, \text{MYCOL}, iacol, \text{NPCOL}), \]
\[ \text{LOCc } (ja+n-1) = \text{numroc}(ja+n-1, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}), \]
\[ \text{and numroc, indxg2p are ScaLAPACK tool functions.} \]

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfo function.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pXerbla.

Workspace array of size \( lrwork \) (complex flavors only).

( local or global) size of \( rwork \) (complex flavors only). The value of \( lrwork \) must be at least
\[ lwork \geq \text{LOCc } (ja+n-1) + nq_0. \]

Here
\[ iroff = \text{mod}(ia-1, mb_a), \quad icoff = \text{mod}(ja-1, nb_a), \]
\[ iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrc_a, \text{NPROW}), \]
\[ iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}), \]
\[ mp_0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}), \]
\[ nq_0 = \text{numroc}(n+icooff, nb_a, \text{MYCOL}, iacol, \text{NPCOL}), \]
\[ \text{LOCc } (ja+n-1) = \text{numroc}(ja+n-1, nb_a, \text{MYCOL}, csrc_a, \text{NPCOL}), \]
\[ \text{and numroc, indxg2p are ScaLAPACK tool functions.} \]

You can determine MYROW, MYCOL, NPROW and NPCOL by calling the blacs_gridinfo function.

If \( lrwork = -1 \), then \( lrwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pXerbla.
Output Parameters

The elements on and above the diagonal of \( \text{sub}(A) \) contain the \( \min(m,n) \)-by-\( n \) upper trapezoidal matrix \( R \) (\( R \) is upper triangular if \( m \geq n \)); the elements below the diagonal, with the array \( \text{tau} \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\( \text{ipiv} \)

(local) Array of size \( \text{LOCc}(ja+n-1) \).

\( \text{ipiv}[i] = k \), the local \((i+1)\)-th column of \( \text{sub}(A)*P \) was the global \( k \)-th column of \( \text{sub}(A) \) \((0 \leq i < \text{LOCc}(ja+n-1)) \). \( \text{ipiv} \) is tied to the distributed matrix \( A \).

\( \text{tau} \)

(local)

Array of size \( \text{LOCc}(ja+\min(m, n)-1) \).

Contains the scalar factor \( \text{tau} \) of elementary reflectors. \( \text{tau} \) is tied to the distributed matrix \( A \).

\( \text{work}[0] \)

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{rwork}[0] \)

On exit, \( \text{rwork}[0] \) contains the minimum value of \( lrwork \) required for optimum performance.

\( \text{info} \)

(global)

\( = 0 \), the execution is successful.

\( < 0 \), if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors

\[ Q = H(1)*H(2)*...*H(k) \]

where \( k = \min(m,n) \).

Each \( H(i) \) has the form

\[ H = I - \tau \cdot v \cdot v' \]

where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(1:i-1) = 0 \) and \( v(i) = 1 \); \( v(i+1:m) \) is stored on exit in \( A(ia+i:ia+m-1, ja+i-1) \).

The matrix \( P \) is represented in \( \text{ipiv} \) as follows: if \( \text{ipiv}[j] = i \) then the \((j+1)\)-th column of \( P \) is the \( i \)-th canonical unit vector \((0 \leq j < \text{LOCc}(ja+n-1)) \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{porgqr}

Generates the orthogonal matrix \( Q \) of the QR factorization formed by \texttt{pgeqrf}.
Syntax

```c
void psorgqr (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorgqr (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
```

Include Files
- mkl_scalapack.h

Description
The `p?orgqr` function generates the whole or part of \( m \)-by-\( n \) real distributed matrix \( Q \) denoting \( A(ia:ia+m-1, ja:ja+n-1) \) with orthonormal columns, which is defined as the first \( n \) columns of a product of \( k \) elementary reflectors of order \( m \) \( Q = H(1)*H(2)*...*H(k) \) as returned by `p?geqrf`.

Input Parameters
- \( m \) (global) The number of rows in the matrix sub(\( Q \)) \((m \geq 0)\).
- \( n \) (global) The number of columns in the matrix sub(\( Q \)) \((n \geq 0)\).
- \( k \) (global) The number of elementary reflectors whose product defines the matrix \( Q(n \geq k \geq 0) \).
- \( a \) (local) Pointer into the local memory to an array of local size \( ld_a*LOCc(ja+n-1) \). The \( j \)-th column of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(j), ja \leq j \leq ja+k-1 \), as returned by `p?geqrf` in the \( k \) columns of its distributed matrix argument \( A(ia:* \), ja:ja+k-1).\)
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.
- \( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).
- \( tau \) (local) Array of size \( LOCc(ja+k-1) \). Contains the scalar factor \( tau[j] \) of elementary reflectors \( H(j+1) \) as returned by `p?geqrf` \( 0 \leq j < LOCc(ja+k-1) \). \( tau \) is tied to the distributed matrix \( A \).
- \( work \) (local) Workspace array of size of \( lwork \).
- \( lwork \) (local or global) size of \( work \). Must be at least \( lwork \geq nb_a*(nqa0 + mpa0 + nb_a) \), where...
irow = indxg2p(ia, mb_a, MYROW, rsra, NPROW),
ja = indxg2p(ja, nb_a, MYCOL, csra, NPCOL),
mpa0 = numroc(m+irow, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+ja, nb_a, MYCOL, iacol, NPCOL);
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a
Contains the local pieces of the m-by-n distributed matrix Q.

work[0]
On exit, [0] contains the minimum value of lwork required for optimum performance.

info
(global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

pzungqr
Generates the complex unitary matrix Q of the QR factorization formed by p?geqr.

Syntax
void pcungqr (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info );
void pzungqr (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info );

Include Files
• mkl_scalapack.h

Description
This function generates the whole or part of m-by-n complex distributed matrix Q denoting A(ia:ia+m-1, ja:ja+n-1) with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m

$$Q = H(1)*H(2)*...*H(k)$$
as returned by `p?geqrf`.

**Input Parameters**

- **m** (global) The number of rows in the matrix sub(Q); \(m \geq 0\).
- **n** (global) The number of columns in the matrix sub(Q) \(m \geq n \geq 0\).
- **k** (global) The number of elementary reflectors whose product defines the matrix Q \(n \geq k \geq 0\).
- **a** (local) Pointer into the local memory to an array of size \(lld_a \times \text{LOCc}(ja+n-1)\). The \(j\)-th column of the matrix stored in \(a\) must contain the vector that defines the elementary reflector \(H(j)\), \(ja \leq j \leq ja + k - 1\), as returned by `p?geqrf` in the \(k\) columns of its distributed matrix argument \(A(ia:*, ja:ja+k-1)\).
- **ia, ja** (global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.
- **desca** (global and local) array of size \(dlen_\_\). The array descriptor for the distributed matrix \(A\).
- **tau** (local) Array of size \(\text{LOCc}(ja+k-1)\). Contains the scalar factor \(\tau[j]\) of elementary reflectors \(H(j+1)\) as returned by `p?geqrf` \(0 \leq j < \text{LOCc}(ja+k-1)\). \(\tau\) is tied to the distributed matrix \(A\).
- **work** (local) Workspace array of size of \(lwork\).
- **lwork** (local or global) size of \(work\), must be at least \(lwork \geq nb_a \times (nqa0 + mpa0 + nb_a)\), where
- \(\text{ipoffa} = \text{mod}(ia-1, mb_a)\),
- \(\text{icoffa} = \text{mod}(ja-1, nb_a)\),
- \(\text{iarow} = \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsra}, \text{NPROW})\),
- \(\text{iacol} = \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csra}, \text{NPCOL})\),
- \(mpa0 = \text{numroc}(m+i\text{ipoffa}, mb_a, \text{MYROW}, \text{iarow}, \text{NPROW})\),
- \(nqa0 = \text{numroc}(n+i\text{icoffa}, nb_a, \text{MYCOL}, \text{iacol}, \text{NPCOL})\)

`indxg2p` and `numroc` are ScaLAPACK tool functions; `MYROW`, `MYCOL`, `NPROW` and `NPCOL` can be determined by calling the function `blacs_gridinfo`.

If \(lwork = -1\), then \(lwork\) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pXerbla`.

**Output Parameters**

- **a** Contains the local pieces of the \(m\)-by-\(n\) distributed matrix \(Q\).
work[0]

On exit work[0] contains the minimum value of lwork required for optimum performance.

info

(global)

= 0: the execution is successful.

< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormqr

Multiplies a general matrix by the orthogonal matrix Q of the QR factorization formed by p?geqrf.

Syntax

void psormqr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);

void pdormqr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

The p?ormqr function overwrites the general real m-by-n distributed matrix sub (C) = C(ic:ic+m-1,ja:ja+n-1) with

\[
\begin{align*}
\text{side} &= 'L' & \quad & \text{side} &= 'R' \\
\text{trans} &= 'N': & Q*\text{sub}(C) & \quad & \text{sub}(C)*Q \\
\text{trans} &= 'T': & Q^T*\text{sub}(C) & \quad & \text{sub}(C)*Q^T
\end{align*}
\]

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors

\[Q = H(1)H(2)...H(k)\]

as returned by p?geqrf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters

side

(global)

='L': Q or Q^T is applied from the left.

='R': Q or Q^T is applied from the right.

trans

(global)

='N', no transpose, Q is applied.

='T', transpose, Q^T is applied.
(global) The number of rows in the distributed matrix sub(C) \((m \geq 0)\).

\(n\) (global) The number of columns in the distributed matrix sub(C) \((n \geq 0)\).

\(k\) (global) The number of elementary reflectors whose product defines the matrix \(Q\). Constraints:

If \(side = 'L'\), \(m \geq k \geq 0\)

If \(side = 'R'\), \(n \geq k \geq 0\).

\(a\) (local)

Pointer into the local memory to an array of size \(lld_a*LOCc(ja+n-1)\). The \(j\)-th column of the matrix stored in \(a\) must contain the vector that defines the elementary reflector \(H(j)\), \(ja \leq j \leq ja+k-1\), as returned by \(p?geqrf\) in the \(k\) columns of its distributed matrix argument \(A(ia:*, ja:ja+k-1)\). \(A(ia:*, ja:ja+k-1)\) is modified by the function but restored on exit.

If \(side = 'L'\), \(lld_a \geq \max(1, LOCr(ia+m-1))\)

If \(side = 'R'\), \(lld_a \geq \max(1, LOCr(ia+n-1))\)

\(ia, ja\) (global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.

\(desca\) (global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(A\).

\(tau\) (local)

Array of size \(LOCc(ja+k-1)\).

Contains the scalar factor \(tau[j]\) of elementary reflectors \(H(j+1)\) as returned by \(p?geqrf\) \((0 \leq j < LOCc(ja+k-1))\). \(tau\) is tied to the distributed matrix \(A\).

\(c\) (local)

Pointer into the local memory to an array of local size \(lld_c*LOCc(jc+n-1)\).

Contains the local pieces of the distributed matrix sub(C) to be factored.

\(ic, jc\) (global) The row and column indices in the global matrix \(C\) indicating the first row and the first column of the matrix sub(C), respectively.

\(desc\) (global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(C\).

\(work\) (local)

Workspace array of size of \(lwork\).

\(lwork\) (local or global) size of \(work\), must be at least:

if \(side = 'L'\),

\[lwork \geq \max((nb_a*(nb_a-1))/2, (nqc0+mpc0)*nb_a) + nb_a*nb_a\]

else if \(side = 'R'\),

\[lwork \geq \max((nb_a*(nb_a-1))/2, (nqc0+\max(npa0+\text{numroc}(\text{numroc}(n+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmq), mpc0))*nb_a) + nb_a*nb_a\]
end if 
where

\[ \text{lcmq} = \frac{\text{lcm}}{\text{NPCOL}} \quad \text{with} \quad \text{lcm} = \text{ilcm} (\text{NPROW}, \text{NPCOL}), \]
\[ \text{iroffa} = \text{mod}(\text{ia}-1, \text{mb}_a), \]
\[ \text{icoffa} = \text{mod}(\text{ja}-1, \text{nb}_a), \]
\[ \text{iarow} = \text{indxg2p}(\text{ia}, \text{mb}_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}), \]
\[ \text{npa0} = \text{numroc}(\text{n}+\text{iroffa}, \text{mb}_a, \text{MYROW}, \text{iarow}, \text{NPROW}), \]
\[ \text{iroffc} = \text{mod}(\text{ic}-1, \text{mb}_c), \]
\[ \text{icoffc} = \text{mod}(\text{jc}-1, \text{nb}_c), \]
\[ \text{icrow} = \text{indxg2p}(\text{ic}, \text{mb}_c, \text{MYROW}, \text{rsrc}_c, \text{NPROW}), \]
\[ \text{iccol} = \text{indxg2p}(\text{jc}, \text{nb}_c, \text{MYCOL}, \text{csrc}_c, \text{NPCOL}), \]
\[ \text{mpc0} = \text{numroc}(\text{m}+\text{iroffc}, \text{mb}_c, \text{MYROW}, \text{icrow}, \text{NPCOL}), \]
\[ \text{nqc0} = \text{numroc}(\text{n}+\text{icoffc}, \text{nb}_c, \text{MYCOL}, \text{iccol}, \text{NPCOL}), \]
\[ \text{ilcm}, \text{indxg2p} \quad \text{and} \quad \text{numroc} \quad \text{are ScaLAPACK tool functions}; \quad \text{MYROW}, \text{MYCOL}, \]
\[ \text{NPROW} \quad \text{and} \quad \text{NPCOL} \quad \text{can be determined by calling the function} \]
\[ \text{blacs_gridinfo}. \]

If \( \text{lwork} = -1 \), then \( \text{lwork} \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

\section*{Output Parameters}

\subsection*{c}
Overwritten by the product \( Q\*\text{sub}(C) \), or \( Q^T\*\text{sub}(C) \), or \( \text{sub}(C)*Q^T \), or \( \text{sub}(C)*Q \).

\subsection*{work[0]}
On exit \( \text{work[0]} \) contains the minimum value of \( \text{lwork} \) required for optimum performance.

\subsection*{info}
\begin{itemize}
  \item \( \text{info} = 0 \): the execution is successful.
  \item \( \text{info} < 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i\*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).
\end{itemize}

\section*{See Also}

\subsection*{Overview}
for details of ScaLAPACK array descriptor structures and related notations.

\subsection*{p?unmqr}
\textit{Multiplies a complex matrix by the unitary matrix \( Q \) of the QR factorization formed by p?geqrf.}
Syntax

void pcunmqr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau,
MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, MKL_Complex8 *work,
MKL_INT *lwork, MKL_INT *info);

void pzunmqr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau,
MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, MKL_Complex16 *work,
MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
This function overwrites the general complex m-by-n distributed matrix sub(C) = C(ic:ic+m-1,ic:ic+n-1)
with

side = 'L': Q or QH is applied from the left.
side = 'R': Q or QH is applied from the right.

trans = 'N': Q*sub(C)
trans = 'C': QH*sub(C)

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors
Q = H(1)H(2)...H(k) as returned by p?geqr3. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters

side
(global)
= 'L': Q or QH is applied from the left.
= 'R': Q or QH is applied from the right.

trans
(global)
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, QH is applied.

m
(global) The number of rows in the distributed matrix sub(C) (m≥0).

n
(global) The number of columns in the distributed matrix sub(C) (n≥0).

k
(global) The number of elementary reflectors whose product defines the matrix Q. Constraints:
If side = 'L', m≥k≥0
If side = 'R', n≥k≥0.

a
(local)
Pointer into the local memory to an array of size llda*locC(ja+k-1). The j-th column of the matrix stored in a must contain the vector that defines the elementary reflector H(j), ja≤j≤ja+k-1, as returned by p?geqr3 in the k columns of its distributed matrix argument A(ia:*, ja:ja+k-1). A(ia:*, ja:ja+k-1) is modified by the function but restored on exit.
If $side = 'L'$, $lld_a \geq \max(1, \text{LOCr}(ia+m-1))$

If $side = 'R'$, $lld_a \geq \max(1, \text{LOCr}(ia+n-1))$

$ia, ja$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

$desca$ (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$tau$ (local)
Array of size $\text{LOCc}(ja+k-1)$.
Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by $\text{p?geqr2}$ ($0 \leq j < \text{LOCc}(ja+k-1)$). $\tau$ is tied to the distributed matrix $A$.

c (local)
Pointer into the local memory to an array of local size $lld_c*\text{LOCc}(jc+n-1)$.
Contains the local pieces of the distributed matrix sub($C$) to be factored.

$ic, jc$ (global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix $C$, respectively.

$descc$ (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $C$.

$work$ (local)
Workspace array of size of $lwork$.

$lwork$ (local or global) size of $work$, must be at least:
If $side = 'L'$,
$$lwork \geq \max\left(\frac{nb_a*(nb_a-1)}{2}, (nqc0 + mpc0)*nb_a + nb_a*nb_a\right)$$
else if $side = 'R'$,
$$lwork \geq \max\left(\frac{nb_a*(nb_a-1)}{2}, (nqc0 + \max(npa0 + \text{numroc}(\text{numroc}(n+iroffa, mb_a, 0, 0, \text{NPCOL}, nb_a, 0, 0, \text{lcmq}), mpc0)))*nb_a + nb_a*nb_a\right)$$
end if

where
$$\text{lcmq} = \text{lcm}/\text{NPCOL} \text{ with lcm = ilcm (NPROW, NPCOL)},$$
iroffa = mod($ia-1, mb_a$),
icoffa = mod($ja-1, nb_a$),
$$iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}),$$
npa0 = $\text{numroc}(n+iroffa, mb_a, \text{MYROW}, iarow, \text{NPROW})$,
iroffc = mod($ic-1, mb_c$),
icoffc = mod($jc-1, nb_c$),
icrow = $\text{indxg2p}(ic, mb_c, \text{MYROW}, \text{rsrc}_c, \text{NPROW})$,
iccol = $\text{indxg2p}(jc, nb_c, \text{MYCOL}, \text{csrc}_c, \text{NPCOL})$. 
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
Overwritten by the product Q*sub(C), or Q^H*sub(C), or sub(C)*Q^H, or sub(C)*Q.

work[0]
On exit work[0] contains the minimum value of lwork required for optimum performance.

info
(global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gelqf
Computes the LQ factorization of a general rectangular matrix.

Syntax

void psgelqf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgelqf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgelqf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgelqf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

The p?gelqf function computes the LQ factorization of a real/complex distributed m-by-n matrix sub(A) = A(ia:ia+m-1,ja:ja+n-1) = L*Q.
Input Parameters

\( m \) (global) The number of rows in the distributed submatrix \( \text{sub}(A) \) \((m \geq 0)\).

\( n \) (global) The number of columns in the distributed submatrix \( \text{sub}(A) \) \((n \geq 0)\).

\( a \) (local)

Pointer into the local memory to an array of local size \( \text{lld}_a \cdot \text{LOCc}(ja+n-1) \).

Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.

\( ia, ja \) (global) The row and column indices in the global array \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1,ja:ja+n-1) \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \text{work} \) (local)

Workspace array of size of \( lwork \).

\( lwork \) (local or global) size of \( \text{work} \), must be at least \( lwork \geq mb_a \cdot (mp0 + nq0 + mb_a) \), where

\[
\begin{align*}
     & iroff = \text{mod}(ia-1, mb_a), \\
     & icoff = \text{mod}(ja-1, nb_a), \\
     & iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsrc_a}, \text{NPROW}), \\
     & iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csrc_a}, \text{NPCOL}), \\
     & mp0 = \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}), \\
     & nq0 = \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})
\end{align*}
\]

\text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.}

**NOTE**

\( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\( a \)

The elements on and below the diagonal of \( \text{sub}(A) \) contain the \( m \)-by-\( \min(m,n) \) lower trapezoidal matrix \( L \) (\( L \) is lower trapezoidal if \( m \leq n \)); the elements above the diagonal, with the array \( \tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\( \tau \) (local)
Array of size $\text{LOCr}(ia+\min(m,n)-1)$.

Contains the scalar factors of elementary reflectors. $\tau$ is tied to the distributed matrix $A$.

$\text{work}[0]$

On exit, $\text{work}[0]$ contains the minimum value of $l\text{work}$ required for optimum performance.

$info$

(global)

$= 0$: the execution is successful.

$< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

**Application Notes**

The matrix $Q$ is represented as a product of elementary reflectors

$$Q = H^{(ia+k-1)}*H^{(ia+k-2)}*...*H^{(ia)},$$

where $k = \min(m,n)$

Each $H(i)$ has the form

$$H(i) = I - \tau*V*V'$$

where $\tau$ is a real/complex scalar, and $V$ is a real/complex vector with $V(1:i-1) = 0$ and $V(i) = 1$; $V(i+1:n)$ is stored on exit in $A(ia+i-1,ja+i:ja+n-1)$, and $\tau$ in $\text{tau}[ia+i-2]$.

**See Also**

*Overview* for details of ScaLAPACK array descriptor structures and related notations.

**porglq**

Generates the real orthogonal matrix $Q$ of the $LQ$ factorization formed by $\text{pgelqf}$.

**Syntax**

```c
void psorglq (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorglq (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The $\text{porglq}$ function generates the whole or part of $m$-by-$n$ real distributed matrix $Q$ denoting $A(ia:ia + m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first $m$ rows of a product of $k$ elementary reflectors of order $n$

$$Q = H(k)*...*H(2)*H(1)$$

as returned by $\text{pgelqf}$.
Input Parameters

- \( m \) (global) The number of rows in the matrix sub(\( Q \)); \( m \geq 0 \).
- \( n \) (global) The number of columns in the matrix sub(\( Q \)); \( n \geq m \geq 0 \).
- \( k \) (global) The number of elementary reflectors whose product defines the matrix \( Q \); \( m \geq k \geq 0 \).
- \( a \) (local) Pointer into the local memory to an array of local size \( lld_a \cdot LOCc(ja+n-1) \).
  On entry, the \( i \)-th row of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia \leq i \leq ia+k-1 \), as returned by \( p?gelqf \) in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1, ja:*) \).
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.
- \( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).
- \( work \) (local) Workspace array of size of \( lwork \).
- \( lwork \) (local or global) size of \( work \), must be at least
  \[ lwork \geq mb_a \cdot (mpa0+nqa0+mb_a) \], where
  \[ iroffa = \text{mod}(ia-1, mb_a), \]
  \[ icoffa = \text{mod}(ja-1, nb_a), \]
  \[ iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrch_a, NPROW), \]
  \[ iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrch_a, NPCOL), \]
  \[ mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW), \]
  \[ nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL) \]

NOTE \( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

\text{indxg2p} and \text{numroc} are ScaLAPACK tool functions; \( \text{MYROW}, \text{MYCOL}, \text{NPROW} \) and \( \text{NPCOL} \) can be determined by calling the function \text{blacs_gridinfo}.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

- \( a \) Contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \) to be factored.
- \( tau \) (local)
Array of size $\text{LOCr}(ia+k-1)$.

Contains the scalar factors $\tau[j]$ of elementary reflectors $H(j+1)$, $0 \leq j < \text{LOCr}(ia+k-1)$. $\tau$ is tied to the distributed matrix $A$.

$\text{work}[0]$

On exit, $\text{work}[0]$ contains the minimum value of $l\text{work}$ required for optimum performance.

$info$

(global)

$ = 0$: the execution is successful.

$ < 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

$p\text{?unglq}$

Generates the unitary matrix $Q$ of the LQ factorization formed by $p\text{?gelqf}$.

Syntax

void pcunglq (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzunglq (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

This function generates the whole or part of $m$-by-$n$ complex distributed matrix $Q$ denoting $A(ia:ia+n-1,ja:ja+n-1)$ with orthonormal rows, which is defined as the first $m$ rows of a product of $k$ elementary reflectors of order $n$

$Q = (H(k))^H*...*(H(2))^H*(H(1))^H$ as returned by $p\text{?gelqf}$.

Input Parameters

$m$

(global) The number of rows in the matrix $\text{sub}(Q)$ ($m \geq 0$).

$n$

(global) The number of columns in the matrix $\text{sub}(Q)$ ($n \geq m \geq 0$).

$k$

(global) The number of elementary reflectors whose product defines the matrix $Q$ ($m \geq k \geq 0$).

$a$

(local)
Pointer into the local memory to an array of local size \( lld_a * \text{LOCc}(ja+n-1) \). On entry, the \( i \)-th row of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia \leq i \leq ia+k-1 \), as returned by \( \text{p?gelqf} \) in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1, ja:*). \)

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \text{tau} \) (local)

Array of size \( \text{LOCr}(ia+k-1) \).

Contains the scalar factors \( \text{tau}[j] \) of elementary reflectors \( H(j+1) \), \( 0 \leq j < \text{LOCr}(ia+k-1) \). \( \text{tau} \) is tied to the distributed matrix \( A \).

\( \text{work} \) (local)

Workspace array of size of \( lwork \).

\( lwork \) (local or global) size of \( \text{work} \), must be at least

\[ lwork \geq mb_a * (mpa0 + nqa0 + mb_a) \]

where

\[
\begin{align*}
iroff &= \text{mod}(ia-1, mb_a), \\
icoff &= \text{mod}(ja-1, nb_a), \\
ivarow &= \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}), \\
ivicol &= \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}), \\
mpa0 &= \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarrow, \text{NPROW}), \\
nqa0 &= \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})
\end{align*}
\]

\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( \text{MYROW} \), \( \text{MYCOL} \), \( \text{NPROW} \) and \( \text{NPCOL} \) can be determined by calling the function \( \text{blacs_gridinfo} \).

**NOTE**

\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \).

### Output Parameters

\( a \)

Contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \) to be factored.

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{info} \)

(gLOBAL)

\( = 0 \): the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had
an illegal value, then info = -(i*100+j); if the i-th argument is a scalar
and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?ormlq
Multiplies a general matrix by the orthogonal matrix Q
of the LQ factorization formed by p?gelqf.

Syntax
void psormlq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float
*a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic,
MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);

void pdormlq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double
*a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT
*ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?ormlq function overwrites the general real m-by-n distributed matrix sub(C) = C(icc:ic+m-1,jc:jc
+n-1) with

side = 'L'
side = 'R'

trans = 'N': Q*sub(C) sub(C)*Q
trans = 'T': Q*T*sub(C) sub(C)*Q*T

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors
Q = H(k)...H(2) H(1)
as returned by p?gelqf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters
side (global)
= 'L': Q or Q*T is applied from the left.
= 'R': Q or Q*T is applied from the right.

trans (global)
= 'N', no transpose, Q is applied.
= 'T', transpose, Q*T is applied.

m (global) The number of rows in the distributed matrix sub(C) (m≥0).
n (global) The number of columns in the distributed matrix sub(C) (n≥0).
k (global) The number of elementary reflectors whose product defines the
matrix Q. Constraints:
If $side = 'L'$, $m \geq k \geq 0$

If $side = 'R'$, $n \geq k \geq 0$.

a (local)

Pointer into the local memory to an array of size $lld_a*\text{LOCc}(ja+m-1)$, if $side = 'L'$ and $lld_a*\text{LOCc}(ja+n-1)$, if $side = 'R'$. The $i$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(i)$, $ia \leq ia+k-1$, as returned by $p?gelqf$ in the $k$ rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

$A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.

$ia, ja$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

desca (global and local) array of size $dlen_\text{a}$. The array descriptor for the distributed matrix $A$.

tau (local)

Array of size $\text{LOCc}(ja+k-1)$.

Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by $p?gelqf$ ($0 \leq j < \text{LOCc}(ja+k-1)$). $\tau$ is tied to the distributed matrix $A$.

c (local)

Pointer into the local memory to an array of local size $lld_c*\text{LOCc}(jc+n-1)$.

Contains the local pieces of the distributed matrix $\text{sub}(C)$ to be factored.

$ic, jc$ (global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix $C$, respectively.

descc (global and local) array of size $dlen_\text{c}$. The array descriptor for the distributed matrix $C$.

work (local)

Workspace array of size of $lwork$.

$lwork$ (local or global) size of the array $work$; must be at least:

If $side = 'L'$,

$$lwork \geq \max((mb_a (mb_a-1))/2, (mpc0 + maxmqa0) + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, \text{NPROW}), mb_a, 0, 0, \text{lcmp}), nqc0) \times mb_a) + mb_a \times mb_a$$

else if $side = 'R'$,

$$lwork \geq \max((mb_a (mb_a-1))/2, (mpc0 + nqc0) \times mb_a + mb_a \times mb_a$$

end if

where

$$lcmp = \text{lcm}/\text{NPROW} \text{ with } \text{lcm} = \text{ilcm(\text{NPROW}, NPCOL)},$$

$$iroffa = \text{mod(ia-1, mb_a)},$$

$$icoffa = \text{mod(ja-1, mb_a)},$$
iacol = indxg2p(ja, nb_a, MYCOL, csr_a, NPCOL),
mqa0 = numroc(m+icoffa, nb_a, MYCOL, iacol, NPCOL),
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsr_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csr_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

NOTE
mod(x,y) is the integer remainder of x/y.

Ilcm, indxg2p and numroc are ScALAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
Overwritten by the product Q*sub(C), or Q'*sub(C), or sub(C)*Q', or sub(C)*Q

work[0]
On exit work[0] contains the minimum value of lwork required for optimum performance.

info
(global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScALAPACK array descriptor structures and related notations.

p?unmlq
Multiplies a general matrix by the unitary matrix Q of the LQ factorization formed by p?gelqf.

Syntax
void pcunmlq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau,
MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *desc, MKL_Complex8 *work,
MKL_INT *lwork, MKL_INT *info );
void pzunmlq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau,
MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work,
MKL_INT *lwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
This function overwrites the general complex m-by-n distributed matrix sub(C) = C(ic:ic+m-1,jc:jc+n-1)
with

side = 'L':
  trans = 'N':  Q*sub(C)
  trans = 'C':  QH*sub(C)
side = 'R':
  trans = 'N':  sub(C)*Q
  trans = 'C':  sub(C)*QH

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors
Q = H(k)' ... H(2)' H(1)'
as returned by p?gelqf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters
side
  (global)
  = 'L': Q or QH is applied from the left.
  = 'R': Q or QH is applied from the right.

trans
  (global)
  = 'N', no transpose, Q is applied.
  = 'C', conjugate transpose, QH is applied.

m
  (global) The number of rows in the distributed matrix sub(C) (m≥0).

n
  (global) The number of columns in the distributed matrix sub(C) (n≥0).

k
  (global) The number of elementary reflectors whose product defines the
  matrix Q. Constraints:
  If side = 'L', m≥k≥0
  If side = 'R', n≥k≥0.

a
  (local)
  Pointer into the local memory to an array of size lld_a*LOCc(ja+m-1), if
  side = 'L' and lld_a*LOCc(ja+n-1), if side = 'R', where lld_a≥
  max(1, LOCr (ia+k-1)). The i-th column of the matrix stored in a
  must contain the vector that defines the elementary reflector H(i),
  ia≤i≤ia+k-1, as returned by p?gelqf in the k rows of its distributed matrix argument
  A(ia:ia+k-1, ja:()). A(ia:ia+k-1, ja:) is modified by the function but
  restored on exit.

ia, ja
  (global) The row and column indices in the global matrix A indicating the
  first row and the first column of the submatrix A, respectively.
desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

tau (local)
Array of size LOCc(ia+k-1).
Contains the scalar factor $\tau[j]$ of elementary reflectors $H(j+1)$ as returned by $p?gelqf$ ($0 \leq j < LOCc(ia+k-1)$). tau is tied to the distributed matrix A.

c (local)
Pointer into the local memory to an array of local size $lld_c*LOCc(jc+n-1)$.
Contains the local pieces of the distributed matrix sub($C$) to be factored.

ic, jc (global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix $C$, respectively.

desccc (global and local) array of size dlen_. The array descriptor for the distributed matrix $C$.

work (local)
Workspace array of size of lwork.

lwork (local or global) size of the array work; must be at least:
If $side = 'L'$,
$$lwork \geq \max((mb_a*(mb_a-1))/2, (mpc0 + maxmqa0) + \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, \text{NPROW}), mb_a, 0, 0, lcmp), nqc0)*mb_a + mb_a*mb_a$$
else if $side = 'R'$,
$$lwork \geq \max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a + mb_a*mb_a$$
end if

where
$$lcmp = lcm/NPROW \text{ with } lcm = \text{lcm (NPROW, NPCOL)}$$
$$iroffa = \text{mod}(ia-1, mb_a)$$
$$icoffa = \text{mod}(ja-1, nb_a)$$
$$iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrca_a, NPCOL)$$
$$mqa0 = \text{numroc}(m + icoffa, nb_a, MYCOL, iacol, NPCOL)$$
$$iroffc = \text{mod}(ic-1, mb_c)$$
$$icoffe = \text{mod}(jc-1, nb_c)$$
$$icrow = \text{indxg2p}(ic, mb_c, MYROW, rsrca_c, NPROW)$$
$$iccol = \text{indxg2p}(jc, nb_c, MYCOL, csrca_c, NPCOL)$$
$$mpc0 = \text{numroc}(m+iroffc, mb_c, MYROW, icrow, NPROW)$$
$$nqc0 = \text{numroc}(n+icoffe, nb_c, MYCOL, iccol, NPCOL)$$
NOTE

mod(x,y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

Overwritten by the product Q*sub(C), or Q'*sub (C), or sub(C)*Q', or sub(C)*Q

work[0]

On exit work[0] contains the minimum value of lwork required for optimum performance.

info (global)

= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geqlf

Computes the QL factorization of a general matrix.

Syntax

void psgeqlf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgeqlf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgeqlf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgeqlf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

• mkl_scalapack.h
Description

The `pgeqlf` function forms the QL factorization of a real/complex distributed \( m \times n \) matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q^*L \).

Input Parameters

- \( m \) (global) The number of rows in the matrix \( \text{sub}(Q) \); \( m \geq 0 \).
- \( n \) (global) The number of columns in the matrix \( \text{sub}(Q) \); \( n \geq 0 \).
- \( a \) (local) Pointer into the local memory to an array of local size \( \text{lld}_a \times \text{LOCc}(ja+n-1) \). Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.
- \( \text{desca} \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( A \).
- \( \text{work} \) (local) Workspace array of size of \( lwork \).
- \( lwork \) (local or global) size of \( \text{work} \), must be at least \( lwork \geq \text{nb}_a \times (mp0 + nq0 + \text{nb}_a) \), where
  - \( iroff = \text{mod}(ia-1, \text{mb}_a) \),
  - \( icoff = \text{mod}(ja-1, \text{nb}_a) \),
  - \( iarow = \text{indxg2p}(ia, \text{mb}_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}) \),
  - \( iacol = \text{indxg2p}(ja, \text{nb}_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}) \),
  - \( mp0 = \text{numroc}(m+iroff, \text{mb}_a, \text{MYROW}, iarow, \text{NPROW}) \),
  - \( nq0 = \text{numroc}(n+icoff, \text{nb}_a, \text{MYCOL}, iacol, \text{NPCOL}) \).

**NOTE**

\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

\( \text{numroc} \) and \( \text{indxg2p} \) are ScaLAPACK tool functions; \( \text{MYROW}, \text{MYCOL}, \text{NPROW} \) and \( \text{NPCOL} \) can be determined by calling the function \( \text{blacs_gridinfo} \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \).

Output Parameters

- \( a \) On exit, if \( m \geq n \), the lower triangle of the distributed submatrix \( A(ia+m-n:ia+m-1, ja:ja+n-1) \) contains the \( n \times n \) lower triangular matrix \( L \); if \( m \leq n \), the elements on and below the \( (n-m) \)-th superdiagonal contain the \( m \times n \)
lower trapezoidal matrix \( L \); the remaining elements, with the array \( \tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\[
\tau \\
\text{(local)}
\]

Array of size \( \text{LOCc}(ja+n-1) \).

Contains the scalar factors of elementary reflectors. \( \tau \) is tied to the distributed matrix \( A \).

\[
\text{work}[0]
\]

On exit, \( \text{work}[0] \) contains the minimum value of \( l_{\text{work}} \) required for optimum performance.

\[
\text{info}
\]

(global)

= 0: the execution is successful.

< 0: if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j); \) if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i. \)

Application Notes
The matrix \( Q \) is represented as a product of elementary reflectors
\[
Q = H(ja+k-1)*...*H(ja+1)*H(ja)
\]
where \( k = \min(m,n) \)

Each \( H(i) \) has the form
\[
H(i) = I - \tau_{i}v_{i}v'_{i}
\]
where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(m-k+i+1:m) = 0 \) and \( v(m-k+i) = 1; v(1:m-k+i-1) \) is stored on exit in \( A(ia:ia+m-k+i-2, ja+n-k+i-1) \), and \( \tau \) in \( \tau[ja+n-k+i-2] \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\[
p\text{orgql}
\]
Generates the orthogonal matrix \( Q \) of the QL factorization formed by \( p\text{geqlf} \).

Syntax
\[
\text{void psorgql (MKL_INT } *m, \text{ MKL_INT } *n, \text{ MKL_INT } *k, \text{ float } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ float } *tau, \text{ float } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );
\]
\[
\text{void pdorgql (MKL_INT } *m, \text{ MKL_INT } *n, \text{ MKL_INT } *k, \text{ double } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ double } *tau, \text{ double } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );
\]

Include Files
- mkl_scalapack.h

Description
The \( p\text{orgql} \) function generates the whole or part of \( m\)-by-\( n \) real distributed matrix \( Q \) denoting \( A(ia:ia+m-1, ja:ja+n-1) \) with orthonormal rows, which is defined as the first \( m \) rows of a product of \( k \) elementary reflectors of order \( n \)
\[
Q = H(k)*...*H(2)*H(1)
\]
Input Parameters

$m$  
(global) The number of rows in the matrix sub$(Q)$, $(m \geq 0)$.

$n$  
(global) The number of columns in the matrix sub$(Q)$, $(m \geq n \geq 0)$.

$k$  
(global) The number of elementary reflectors whose product defines the matrix $Q$, $(n \geq k \geq 0)$.

$a$  
(local)  
Pointer into the local memory to an array of local size $lld_a \times LOCc(ja+n-1)$. On entry, the $j$-th column of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(j), ja+n-k \leq j \leq ja+n-1$, as returned by $p?geqlf$ in the $k$ columns of its distributed matrix argument $A(ia:*, ja+n-k:ja+n-1)$.

$ia, ja$  
(global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A(ia:ia+m-1, ja:ja+n-1)$, respectively.

$desca$  
(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$tau$  
(local)  
Array of size $LOCc(ja+n-1)$. Contains the scalar factors $\tau[j]$ of elementary reflectors $H(j+1), 0 \leq j < LOCr(ia+n-1)$. $tau$ is tied to the distributed matrix $A$.

$work$  
(local)  
Workspace array of size of $lwork$.

$lwork$  
(local or global) size of $work$, must be at least $lwork \geq nb_a \times (nqa0 + mpa0 + nb_a)$, where  

$$
iroffa = \text{mod}(ia-1, mb_a),
$$

$$
icoffa = \text{mod}(ja-1, nb_a),
$$

$$
iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW),
$$

$$
iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL),
$$

$$
mpa0 = \text{numroc}(m+iroffa, mb_a, MYROW, iarow, NPROW),
$$

$$
nqa0 = \text{numroc}(n+icoffa, nb_a, MYCOL, iacol, NPCOL)
$$

NOTE

mod$(x, y)$ is the integer remainder of $x/y$.

indxg2p and numroc are ScALAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.
If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \texttt{pxerbla}.

**Output Parameters**

\( a \)

Contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \) to be factored.

\( \text{work}[0] \)

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{info} \) (global)

= 0: the execution is successful.

< 0: if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i\times100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{pzungql}

Generates the unitary matrix \( Q \) of the QL factorization formed by \texttt{p?geqlf}.

**Syntax**

```c
void pcungql (const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *tau , MKL_Complex8 *work , const MKL_INT *lwork , MKL_INT *info );
void pzungql (const MKL_INT *m , const MKL_INT *n , const MKL_INT *k , MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *tau , MKL_Complex16 *work , const MKL_INT *lwork , MKL_INT *info );
```

**Include Files**

- \texttt{mklscalapack.h}

**Description**

This function generates the whole or part of \( m \)-by-\( n \) complex distributed matrix \( Q \) denoting \( A(ia:ia+n-1,ja:ja+n-1) \) with orthonormal rows, which is defined as the first \( n \) columns of a product of \( k \) elementary reflectors of order \( m \)

\[ Q = (H(k))^{H}\star(H(2))^{H}\star(H(1))^{H} \]

as returned by \texttt{p?geqlf}.

**Input Parameters**

\( m \) (global) The number of rows in the matrix sub(\( Q \)) \( (m\geq0) \).

\( n \) (global) The number of columns in the matrix sub(\( Q \)) \( (n\geq0) \).

\( k \) (global) The number of elementary reflectors whose product defines the matrix \( Q \) \( (n\geq k \geq 0) \).
**Output Parameters**

- **a**
  
  Contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \) to be factored.

- **work[0]**

  On exit, `work[0]` contains the minimum value of `lwork` required for optimum performance.

- **info**

  (global)

  \( = 0 \): the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i*100+j) \); if the i-th argument is a scalar and had an illegal value, then \( info = -i \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?ormql}

Multiplies a general matrix by the orthogonal matrix \( Q \) of the QL factorization formed by \textbf{p?geqlf}.

\textbf{Syntax}

```c
void psormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , float *work , MKL_INT *lwork , MKL_INT *info );
void pdormql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , double *work , MKL_INT *lwork , MKL_INT *info );
```

\textbf{Include Files}

- mkl_scalapack.h

\textbf{Description}

The \textbf{p?ormql} function overwrites the general real \( m \)-by-\( n \) distributed matrix \( sub(C) = C(ic:ic+m-1,jc:jc+n-1) \) with

\[
\begin{align*}
\text{side} = \text{'L'}: & \quad Q \text{ or } Q^T \text{ is applied from the left.} \\
\text{trans} = \text{'N'}: & \quad Q^*\text{sub}(C) \\
\text{trans} = \text{'T'}: & \quad Q^T*\text{sub}(C)
\end{align*}
\]

where \( Q \) is a real orthogonal distributed matrix defined as the product of \( k \) elementary reflectors

\[ Q = H(k)' \ldots H(2)' H(1)' \]

as returned by \textbf{p?geqlf}. \( Q \) is of order \( m \) if \text{side} = \text{'L'} and of order \( n \) if \text{side} = \text{'R'}.

\textbf{Input Parameters}

\textbf{side} (global)

- \text{'L'}: \( Q \) or \( Q^T \) is applied from the left.
- \text{'R'}: \( Q \) or \( Q^T \) is applied from the right.

\textbf{trans} (global)

- \text{'N'}: no transpose, \( Q \) is applied.
- \text{'T'}: transpose, \( Q^T \) is applied.

\textbf{m} (global) The number of rows in the distributed matrix \( \text{sub}(C) \), \((m \geq 0)\).

\textbf{n} (global) The number of columns in the distributed matrix \( \text{sub}(C) \), \((n \geq 0)\).

\textbf{k} (global) The number of elementary reflectors whose product defines the matrix \( Q \). Constraints:
If \( \text{side} = 'L' \), \( m \geq k \geq 0 \).

If \( \text{side} = 'R' \), \( n \geq k \geq 0 \).

\( a \)  
(local)

Pointer into the local memory to an array of size \( \text{lld}_a \times \text{LOCc}(ja+k-1) \). The \( j \)-th column of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(j) \), \( ja \leq j \leq ja+k-1 \), as returned by \( \text{p?gelqf} \) in the \( k \) columns of its distributed matrix argument \( A(ia:*, ja:ja+k-1) \). \( A(ia:*, ja:ja+k-1) \) is modified by the function but restored on exit.

If \( \text{side} = 'L' \), \( \text{lld}_a \geq \max(1, \text{LOCr}(ia+m-1)) \).
If \( \text{side} = 'R' \), \( \text{lld}_a \geq \max(1, \text{LOCr}(ia+n-1)) \).

\( \text{ia}, \text{ja} \)  
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( \text{desca} \)  
(global and local) array of size \( dlen_\text{c} \). The array descriptor for the distributed matrix \( A \).

\( \tau \)  
(local)

Array of size \( \text{LOCc}(ja+n-1) \).

Contains the scalar factor \( \tau[j] \) of elementary reflectors \( H(j+1) \) as returned by \( \text{p?geqlf} \) \( (0 \leq j < \text{LOCc}(ja+k-1)) \). \( \tau \) is tied to the distributed matrix \( A \).

\( c \)  
(local)

Pointer into the local memory to an array of local size \( \text{lld}_c \times \text{LOCc}(jc+n-1) \).

Contains the local pieces of the distributed matrix sub(\( C \)) to be factored.

\( \text{ic}, \text{jc} \)  
(global) The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

\( \text{descC} \)  
(global and local) array of size \( dlen_\text{c} \). The array descriptor for the distributed matrix \( C \).

\( \text{work} \)  
(local)

Workspace array of size of \( \text{lwork} \).

\( \text{lwork} \)  
(local or global) dimension of \( \text{work} \), must be at least:

If \( \text{side} = 'L' \),
\[ lwork \geq \max((\text{nb}_a \times (\text{nb}_a-1))/2, (\text{nqc}_0 + \text{mpc}_0) \times \text{nb}_a + \text{nb}_a \times \text{nb}_a) \]
else if \( \text{side} = 'R' \),
\[ lwork \geq \max((\text{nb}_a \times (\text{nb}_a-1))/2, (\text{nqc}_0 + \max(\text{npa}_0 + \text{numroc}(\text{numroc}(\text{npa}_0 + \text{icoffc}, \text{nb}_a, 0, 0, \text{NPCOL}), \text{nb}_a, 0, 0, \text{lcmq}), \text{mpc}_0) \times \text{nb}_a + \text{nb}_a \times \text{nb}_a) \]
end if

where

\( \text{lcmq} = \text{lcm}/\text{NPCOL} \) with \( \text{lcm} = \text{ilcm}(\text{NPROW}, \text{NPCOL}), \text{icoffc} = \text{mod}(\text{ia}-1, \text{mb}_a) \).
icoffa = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsrc_a, NPROW),
npa0= numroc(n + iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(ic-1, mb_c),
icoffc = mod(ic-1, mb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

NOTE

\text{mod}(x,y) \text{ is the integer remainder of } x/y.

\text{lcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.}

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c

Overwritten by the product Q* sub(C), or Q’*sub (C), or sub(C)* Q’, or sub(C)* Q

work[0]

On exit work[0] contains the minimum value of lwork required for optimum performance.

info

(global)

= 0: the execution is successful.

< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmql

\text{Multiplies a general matrix by the unitary matrix } Q \text{ of the } QL \text{ factorization formed by p?geqlf.}

Syntax

void p?unmql (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k ,
MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
void pzunmql (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
This function overwrites the general complex m-by-n distributed matrix sub(C) = C(ic:ic+m-1,jc:jc+n-1) with

\[
\begin{align*}
\text{side} = 'L': & \quad \text{Q or } Q^H \text{ is applied from the left.} \\
\text{side} = 'R': & \quad \text{Q or } Q^H \text{ is applied from the right.} \\
\text{trans} = 'N': & \quad Q\text{sub}(C) \\
\text{trans} = 'C': & \quad Q^H\text{sub}(C)
\end{align*}
\]

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors

\[
Q = H(k)' ... H(2)' H(1')
\]
as returned by p?geqlf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters
side
(global)
= 'L': Q or Q^H is applied from the left.
= 'R': Q or Q^H is applied from the right.

trans
(global)
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, Q^H is applied.

m
(global) The number of rows in the distributed matrix sub(C) (\(m\geq0\)).

n
(global) The number of columns in the distributed matrix sub(C) (\(n\geq0\)).

k
(global) The number of elementary reflectors whose product defines the matrix Q. Constraints:
If side = 'L', \(m\geq k \geq 0\)
If side = 'R', \(n\geq k \geq 0\).

a
(local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+k-1). The j-th column of the matrix stored in a must contain the vector that defines the elementary reflector H(j), \(ja\leq j \leq ja+k-1\), as returned by p?geqlf in the k columns of its distributed matrix argument A(ia:*, ja:ja+k-1). A(ia:*, ja:ja+k-1) is modified by the function but restored on exit.
If side = 'L', lld_a \geq \max(1, LOCr(ia+m-1)),
If side = 'R', lld_a \geq \max(1, LOCr(ia+n-1)).
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

desca

(global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

tau

(local)

Array of size \( \text{LOCc}(\text{ia}+\text{n}-1) \).

Contains the scalar factor \( \tau[j] \) of elementary reflectors \( H(j+1) \) as returned by \( \text{p?geqlf} \) \((0 \leq j < \text{LOCc}(\text{ia}+\text{n}-1)) \). \( \tau \) is tied to the distributed matrix \( A \).

c

(local)

Pointer into the local memory to an array of local size \( \text{lld}_c*\text{LOCc}(\text{jc}+\text{n}-1) \).

Contains the local pieces of the distributed matrix sub\((C)\) to be factored.

ic, jc

(global) The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

descc

(global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( C \).

work

Workspace array of size of \( \text{lwork} \).

lwork

(local or global) size of \( \text{work} \), must be at least:

If \( \text{side} = 'L', \)
\[
\text{lwork} \geq \max( (\text{nb}_a*(\text{nb}_a-1))/2, (\text{nqc}_0+\text{mpc}_0)*\text{nb}_a + \text{nb}_a*\text{nb}_a
\]

else if \( \text{side} = 'R', \)
\[
\text{lwork} \geq \max( (\text{nb}_a*(\text{nb}_a-1))/2, (\text{nqc}_0+\text{mpa}_0) + \text{numroc}(\text{numroc}(n+\text{ioccof}_c, \text{nb}_a, 0, 0, \text{NPCOL}), \text{nb}_a, 0, 0, \text{lcm}_q, \text{mpc}_0))*\text{nb}_a
\]
\[
+ \text{nb}_a*\text{nb}_a
\]

end if

where

\( \text{lcmp} = \text{lcm}/\text{NPCOL} \) with \( \text{lcm} = \text{lcm}(\text{NRW}, \text{NPCOL}), \)

\( \text{iroff}_a = \text{mod}(\text{ia}-1, \text{mb}_a), \)

\( \text{icoff}_a = \text{mod}(\text{ja}-1, \text{nb}_a), \)

\( \text{iarow} = \text{indxg2p}(\text{ia}, \text{mb}_a, \text{MYROW}, \text{rsrc}_a, \text{NRW}), \)

\( \text{mpa}_0 = \text{numroc}(n+\text{ioccof}_c, \text{nb}_a, 0, 0, \text{NPCOL}), \)

\( \text{iroff}_c = \text{mod}(\text{ic}-1, \text{mb}_c), \)

\( \text{icoff}_c = \text{mod}(\text{jc}-1, \text{nb}_c), \)

\( \text{icrow} = \text{indxg2p}(\text{ic}, \text{mb}_c, \text{MYROW}, \text{rsrc}_c, \text{NRW}), \)

\( \text{mpc}_0 = \text{numroc}(m+\text{iroff}_c, \text{mb}_c, \text{MYROW}, \text{icrow}, \text{NRW}), \)

\( \text{nqc}_0 = \text{numroc}(n+\text{iroff}_c, \text{nb}_c, \text{MYCOL}, \text{iccol}, \text{NPCOL}), \)
\[ \text{mod}(x, y) \text{ is the integer remainder of } x/y. \]

\[ \text{ilcm}, \text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.} \]

\[ \text{NOTE} \]
\[ \text{mod}(x, y) \text{ is the integer remainder of } x/y. \]

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \).

**Output Parameters**

\[ c \]
Overwritten by the product \( Q^* \) sub(\( C \)), or \( Q^* \) sub (\( C \)), or sub(\( C \))* \( Q \), or sub(\( C \))* \( Q^* \).

\[ \text{work}[0] \]
On exit \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\[ \text{info} \]
\( \) (global)
\[ = 0: \] the execution is successful.
\[ < 0: \] if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?gerqf**

*Computes the RQ factorization of a general rectangular matrix.*

**Syntax**

\[
\text{void psgerqf (MKL_INT } *m, \text{ MKL_INT } *n, \text{ float } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ float } *tau, \text{ float } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );}
\]

\[
\text{void pdgerqf (MKL_INT } *m, \text{ MKL_INT } *n, \text{ double } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ double } *tau, \text{ double } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );}
\]

\[
\text{void pcgerqf (MKL_INT } *m, \text{ MKL_INT } *n, \text{ MKL_Complex8 } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ MKL_Complex8 } *tau, \text{ MKL_Complex8 } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );}
\]

\[
\text{void pzgerqf (MKL_INT } *m, \text{ MKL_INT } *n, \text{ MKL_Complex16 } *a, \text{ MKL_INT } *ia, \text{ MKL_INT } *ja, \text{ MKL_INT } *desca, \text{ MKL_Complex16 } *tau, \text{ MKL_Complex16 } *work, \text{ MKL_INT } *lwork, \text{ MKL_INT } *info );}
\]
Include Files

- mkl_scalapack.h

Description

The `pgerqf` function forms the QR factorization of a general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) as

\[ A = R \times Q \]

Input Parameters

- \( m \)
  (global) The number of rows in the distributed matrix \( \text{sub}(A) \); \( m \geq 0 \).

- \( n \)
  (global) The number of columns in the distributed matrix \( \text{sub}(A) \); \( n \geq 0 \).

- \( a \)
  (local)
  Pointer into the local memory to an array of local size \( lld_a \times \text{LOCc}(ja+n-1) \).
  Contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be factored.

- \( ia, ja \)
  (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \), respectively.

- \( \text{desca} \)
  (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

- \( \text{work} \)
  (local)
  Workspace array of size \( lwork \).

- \( lwork \)
  (local or global) size of \( \text{work} \), must be at least

\[
\text{lwork} \geq \text{mb}_a \times (\text{mp}_0 + \text{nq}_0 + \text{mb}_a),
\]

where

\[
\text{iroff} = \text{mod}(ia-1, \text{mb}_a),
\]

\[
\text{icoff} = \text{mod}(ja-1, \text{nb}_a),
\]

\[
\text{iarow} = \text{indxg2p}(ia, \text{mb}_a, \text{MYROW}, \text{rsnc}_a, \text{NPROW}),
\]

\[
\text{iacol} = \text{indxg2p}(ja, \text{nb}_a, \text{MYCOL}, \text{csnc}_a, \text{NPCOL}),
\]

\[
\text{mp}_0 = \text{numroc}(m+\text{iroff}, \text{mb}_a, \text{MYROW}, \text{iarow}, \text{NPROW}),
\]

\[
\text{nq}_0 = \text{numroc}(n+\text{icoff}, \text{nb}_a, \text{MYCOL}, \text{iacol}, \text{NPCOL})
\]

and \text{numroc}, \text{indxg2p} are ScaLAPACK tool functions; \text{MYROW}, \text{MYCOL}, \text{NPROW} and \text{NPCOL} can be determined by calling the function \text{blacs_gridinfo}.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}. 

\[ \text{mod}(x,y) \] is the integer remainder of \( x/y \). 

\[ nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL) \text{ and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.} \]

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}. 

\[ \text{mod}(x,y) \] is the integer remainder of \( x/y \). 

\[ nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL) \text{ and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.} \]
Output Parameters

On exit, if \( m \leq n \), the upper triangle of \( A(ia:ia+m-1, ja:ja+n-1) \) contains the \( m \)-by-\( m \) upper triangular matrix \( R \); if \( m > n \), the elements on and above the \( (m - n) \)-th subdiagonal contain the \( m \)-by-\( n \) upper trapezoidal matrix \( R \); the remaining elements, with the array \( \tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\( \tau \) (local)

Array of size \( \text{LOCr}(ia+m-1) \).

Contains the scalar factor of elementary reflectors. \( \tau \) is tied to the distributed matrix \( A \).

\( \text{work}[0] \)

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{info} \) (global)

\( = 0 \), the execution is successful.

\( < 0 \), if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors
\[
Q = H(ia)H(ia+1)\ldots H(ia+k-1),
\]
where \( k = \min(m,n) \).

Each \( H(i) \) has the form
\[
H(i) = I - \tau \cdot v \cdot v^T
\]
where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(n-k+i+1:n) = 0 \) and \( v(n-k+i) = 1 \); \( v(1:n-k+i-1) \) is stored on exit in \( A(ia+m-k+i-1,ja:ja+n-k+i-2) \), and \( \tau \) in \( \tau[ia+m-k+i-2] \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{p?orgrq}

Generates the orthogonal matrix \( Q \) of the RQ factorization formed by \texttt{p?gerqf}.

Syntax

\[
\text{void psorgrq (MKL_INT } \* m, \text{ MKL_INT } \* n, \text{ MKL_INT } \* k, \text{ float } \* a, \text{ MKL_INT } \* ia, \text{ MKL_INT } \* ja, \text{ MKL_INT } \* desca, \text{ float } \* \tau, \text{ float } \* \text{work, MKL_INT } \* lwork, \text{ MKL_INT } \* \text{info });}
\]
\[
\text{void pdorgrq (MKL_INT } \* m, \text{ MKL_INT } \* n, \text{ MKL_INT } \* k, \text{ double } \* a, \text{ MKL_INT } \* ia, \text{ MKL_INT } \* ja, \text{ MKL_INT } \* desca, \text{ double } \* \tau, \text{ double } \* \text{work, MKL_INT } \* \text{lwork, MKL_INT } \* \text{info });}
\]

Include Files

- \texttt{mkl_scalapack.h}
Description
The _?orgqr_ function generates the whole or part of _m_ by _n_ real distributed matrix _Q_ denoting _A__(ia:ia+m-1,ja:ja+n-1) with orthonormal rows that is defined as the last _m_ rows of a product of _k_ elementary reflectors of order _n_

\[ Q = H(1)^*H(2)^*...*H(k) \]

as returned by _?gerqf_.

Input Parameters

- **_m_** (global) The number of rows in the matrix sub(_Q_), \((m \geq 0)\).
- **_n_** (global) The number of columns in the matrix sub(_Q_), \((n \geq m \geq 0)\).
- **_k_** (global) The number of elementary reflectors whose product defines the matrix _Q_ \((m \geq k \geq 0)\).
- **_a_** (local) Pointer into the local memory to an array of local size \(lld_a*\text{LOCc}(ja+n-1)\). The _i_-th row of the matrix stored in _a_ must contain the vector that defines the elementary reflector _H_(\(i\)), \(ia \leq i \leq ia+m-1\), as returned by _?gerqf_ in the _k_ rows of its distributed matrix argument _A__(ia+m-k:ia+m-1,ja:*)
- **_ia, ja_** (global) The row and column indices in the global matrix _A_ indicating the first row and the first column of the submatrix _A_, respectively.
- **_desca_** (global and local) array of size \(dlen_\). The array descriptor for the distributed matrix _A_.
- **_tau_** (local) Array of size \(\text{LOCc}(ja+k-1)\).
  Contains the scalar factor _tau_[_] of elementary reflectors _H_(\(_i+1\)) as returned by _?gerqf_, \(0 \leq i < \text{LOCr}(ja+k-1)\). _tau_ is tied to the distributed matrix _A_.
- **_work_** (local) Workspace array of size of _lwork_.
- **_lwork_** (local or global) size of _work_, must be at least \(lwork \geq mb_a*(mpa0 + nqa0 + mb_a)\), where
  \( iroff_a = \text{mod}(ia-1, mb_a)\),
  \( icoff_a = \text{mod}(ja-1, nb_a)\),
  \( iarow = \text{indxg2p}(ia, mb_a, \text{MYROW}, rsrca, \text{NPROW})\),
  \( iacol = \text{indxg2p}(ja, nb_a, \text{MYCOL}, csrca, \text{NPCOL})\),
  \( mpa0 = \text{numroc}(m+iroff_a, mb_a, \text{MYROW}, iarow, \text{NPROW})\),
  \( nqa0 = \text{numroc}(n+icoff_a, nb_a, \text{MYCOL}, iacol, \text{NPCOL})\)

  _indxg2p_ and _numroc_ are ScaLAPACK tool functions; _MYROW_, _MYCOL_, _NPROW_ and _NPCOL_ can be determined by calling the function _blacs_gridinfo_.

_ScaLAPACK Routines_
NOTE

\( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

- \( a \) contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \).
- \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.
- \( \text{info} \) is an (global) output parameter.

\( \text{info} = 0 \) : the execution is successful.
\( < 0 \) : if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

See Also

Overview for details of \text{ScaLAPACK} array descriptor structures and related notations.

\text{p?ungrq}

Generates the unitary matrix \( Q \) of the RQ factorization formed by \text{p?gerqf}.

Syntax

void pcungrq (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzungrq (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description

This function generates the \( m \)-by-\( n \) complex distributed matrix \( Q \) denoting \( A(ia:ia+m-1,ja:ja+n-1) \) with orthonormal rows, which is defined as the last \( m \) rows of a product of \( k \) elementary reflectors of order \( n \)

\( Q = (H(1))^H* (H(2))^H * ... * (H(k))^H \) as returned by \text{p?gerqf}.

Input Parameters

- \( m \) (global) The number of rows in the matrix sub(\( Q \)); \( m \geq 0 \).
- \( n \) (global) The number of columns in the matrix sub(\( Q \)) \( n \geq m \geq 0 \).
The number of elementary reflectors whose product defines the matrix $Q$ ($m \geq k \geq 0$).

Pointer into the local memory to an array of size $lld_a*LOCc(ja+n-1)$. The $i$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(i)$, $ia+m-k \leq i \leq ia+m-1$, as returned by $p?gerqf$ in the $k$ rows of its distributed matrix argument $A(ia+m-k:ia+m-1, ja*)$.

The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

Array of size $LOCr(ia+m-1)$.

Contains the scalar factor $\tau[i]$ of elementary reflectors $H(i+1)$ as returned by $p?gerqf$, $0 \leq i < LOCr(ia+m-1)$. $\tau$ is tied to the distributed matrix $A$.

Workspace array of size of $lwork$.

(size of work, must be at least $lwork \geq mb_a*(mpa0 + nqa0+mb_a)$, where

\[
\begin{align*}
  iroff &= \text{mod}(ia-1, mb_a), \\
  icoff &= \text{mod}(ja-1, nb_a), \\
  iarow &= \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsrc_a}, \text{NPROW}), \\
  iacol &= \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csrc_a}, \text{NPCOL}), \\
  mpa0 &= \text{numroc}(m+iroff, mb_a, \text{MYROW}, iarow, \text{NPROW}), \\
  nqa0 &= \text{numroc}(n+icoff, nb_a, \text{MYCOL}, iacol, \text{NPCOL})
\end{align*}
\]

\[\text{mod}(x, y)\text{ is the integer remainder of } x/y.\]

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pzerbla.

### Output Parameters

Contains the local pieces of the $m$-by-$n$ distributed matrix $Q$.

On exit $work[0]$ contains the minimum value of $lwork$ required for optimum performance.
info (global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

pormr3
Applies an orthogonal distributed matrix to a general m-by-n distributed matrix.

Syntax
void psormr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n, const MKL_INT* k, const MKL_INT* l, const float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const float* tau, float* c, const MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, float* work, const MKL_INT* lwork, MKL_INT* info);
void pdormr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n, const MKL_INT* k, const MKL_INT* l, const double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const double* tau, double* c, const MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, double* work, const MKL_INT* lwork, MKL_INT* info);

Include Files
• mkl_scalapack.h

Description
pormr3 overwrites the general real m-by-n distributed matrix sub( C ) = C(ic:ic+m-1,jc:jc+n-1) with

<table>
<thead>
<tr>
<th>side = 'L'</th>
<th>side = 'R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>trans = 'N'</td>
<td>Q * sub( C )</td>
</tr>
<tr>
<td>trans = 'T'</td>
<td>Q**T * sub( C )</td>
</tr>
</tbody>
</table>

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors
Q = H(1) H(2) . . . H(k)
as returned by pztzrzf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters
side (global)
= 'L': apply Q or Q**T from the Left;
= 'R': apply Q or Q**T from the Right.

trans (global)
= 'N': No transpose, apply Q;
= 'T': Transpose, apply \( Q^T \).

\( m \) (global)
The number of rows to be operated on i.e the number of rows of the distributed submatrix \( \text{sub}(C) \). \( m >= 0 \).

\( n \) (global)
The number of columns to be operated on i.e the number of columns of the distributed submatrix \( \text{sub}(C) \). \( n >= 0 \).

\( k \) (global)
The number of elementary reflectors whose product defines the matrix \( Q \).
If \( \text{side} = 'L' \), \( m >= k >= 0 \),
if \( \text{side} = 'R' \), \( n >= k >= 0 \).

\( l \) (global)
The columns of the distributed submatrix \( \text{sub}(A) \) containing the meaningful part of the Householder reflectors.
If \( \text{side} = 'L' \), \( m >= l >= 0 \),
if \( \text{side} = 'R' \), \( n >= l >= 0 \).

\( a \) (local)
Pointer into the local memory to an array of size \( \text{lld}_a \times \text{LOCc}(ja+m-1) \) if \( \text{side} = 'L' \), and \( \text{lld}_a \times \text{LOCc}(ja+n-1) \) if \( \text{side} = 'R' \), where \( \text{lld}_a \geq \text{MAX}(1,\text{LOCr}(ia+k-1)) \);
On entry, the \( i \)-th row must contain the vector which defines the elementary reflector \( H(i) \), \( ia <= i <= ia+k-1 \), as returned by \( \text{p?tzrzf} \) in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1,ja:*). \)
\( A(ia:ia+k-1,ja:*)) \) is modified by the routine but restored on exit.

\( ia \) (global)
The row index in the global array \( a \) indicating the first row of \( \text{sub}(A) \).

\( ja \) (global)
The column index in the global array \( a \) indicating the first column of \( \text{sub}(A) \).

desca (global and local)
Array of size \( \text{dlen}_a \).
The array descriptor for the distributed matrix \( A \).

\( \tau \) (local)
Array, size \( \text{LOCc}(ia+k-1) \).
This array contains the scalar factors \( \tau(i) \) of the elementary reflectors \( H(i) \) as returned by \( \text{p?tzrzf} \). \( \tau \) is tied to the distributed matrix \( A \).

\( c \) (local)
Pointer into the local memory to an array of size \( \text{lld}_c \times \text{LOCc}(jc+n-1) \).
On entry, the local pieces of the distributed matrix $C$.

$(\text{global})$

The row index in the global array $c$ indicating the first row of $C$.

$(\text{global})$

The column index in the global array $c$ indicating the first column of $C$.

$(\text{global and local})$

Array of size $dlen_\_.$

The array descriptor for the distributed matrix $C$.

$(\text{local})$

Array, size $(lwork)$

$(\text{local})$

The size of the array $work$.

$lwork$ is local input and must be at least

If $side = 'L'$, $lwork >= MpC0 + \text{MAX}(\text{MAX}(1, NqC0 \text{, numroc($numroc(m + IROFFC, mb_a, 0, 0, NPROW), mb_a, 0, 0, NqC0))})$;

If $side = 'R'$, $lwork >= NqC0 + \text{MAX}(1, MpC0)$;

where $LCMP = LCM / NPROW$

$LCM = ilcm(NPROW, NPCOL)$,

$IROFFC = \text{MOD}(ic-1, mb_c)$,

$ICOFFC = \text{MOD}(jc-1, nb_c)$,

$ICROW = \text{indxg2p}(ic, mb_c, MYROW, rsrcc_c, NPROW)$,

$ICCOL = \text{indxg2p}(jc, nb_c, MYCOL, csrcc_c, NPCOL)$,

$MpC0 = \text{numroc}(m+IROFFC, mb_c, MYROW, ICROW, NPROW)$,

$NqC0 = \text{numroc}(n+ICOFFC, nb_c, MYCOL, ICCOL, NPCOL)$,

$lclm$, $\text{indxg2p}$, and $\text{numroc}$ are ScaLAPACK tool functions;

$MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the subroutine $\text{blacs_gridinfo}$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by $\text{pxerbla}$.

**Output Parameters**

$c$

On exit, $\text{sub}(C)$ is overwritten by $Q*\text{sub}(C)$ or $Q^*\text{sub}(C)$ or $\text{sub}(C)*Q$ or $\text{sub}(C)^*Q$.

$work$

On exit, $work[0]$ returns the minimal and optimal $lwork$.

$info$

$(\text{local})$
= 0: successful exit
< 0: If the $i$-th argument is an array and the $j$-th entry had an illegal value, then $info = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

**Application Notes**

**Alignment requirements**

The distributed submatrices $A(ia:\ast, ja:\ast)$ and $C(ic:ic+m-1, jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

- If $side = 'L$',
  \[ nb_a = mb_c \text{ AND ICOFFA} = IROFFC \]

- If $side = 'R$',
  \[ nb_a = nb_c \text{ AND ICOFFA} = ICOFFC \text{ AND IACOL} = ICCOL \]

**p?unmr3**

Applies an orthogonal distributed matrix to a general $m$-by-$n$ distributed matrix.

**Syntax**

```c
void pcunmr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n, const MKL_INT* k, const MKL_INT* l, const MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_Complex8* tau, MKL_Complex8* c, const MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, MKL_Complex8* work, const MKL_INT* lwork, MKL_INT* info);

void pzunmr3 (const char* side, const char* trans, const MKL_INT* m, const MKL_INT* n, const MKL_INT* k, const MKL_INT* l, const MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_Complex16* tau, MKL_Complex16* c, const MKL_INT* ic, const MKL_INT* jc, const MKL_INT* descc, MKL_Complex16* work, const MKL_INT* lwork, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

p?unmr3 overwrites the general complex $m$-by-$n$ distributed matrix $\text{sub}( C ) = C(ic:ic+m-1, jc:jc+n-1)$ with

- $side = 'L'$
  \[ trans = 'N': Q * \text{sub}( C ) \]
  \[ trans = 'C': Q^H * \text{sub}( C ) \]

- $side = 'R'$
  \[ trans = 'N': \text{sub}( C ) * Q \]
  \[ trans = 'C': \text{sub}( C ) * Q^H \]

where $Q$ is a complex unitary distributed matrix defined as the product of $k$ elementary reflectors

$Q = H(1)' H(2)' \ldots H(k)'$

as returned by p?tzrzf. $Q$ is of order $m$ if $side = 'L$' and of order $n$ if $side = 'R$'.

**Input Parameters**

- $side$ (global)
  - $'L$': apply $Q$ or $Q^H$ from the Left;
= 'R': apply $Q$ or $Q^H$ from the Right.

$trans$
    (global)
  = 'N': No transpose, apply $Q$;
  = 'C': Conjugate transpose, apply $Q^H$.

$m$
    (global)
  The number of rows to be operated on i.e the number of rows of the
distributed submatrix sub$(C)$. $m \geq 0$.

$n$
    (global)
  The number of columns to be operated on i.e the number of columns of the
distributed submatrix sub$(C)$. $n \geq 0$.

$k$
    (global)
  The number of elementary reflectors whose product defines the matrix $Q$.
  If $side = 'L', m \geq k \geq 0$, if $side = 'R', n \geq k \geq 0$.

$l$
    (global)
  The columns of the distributed submatrix sub$(A)$ containing the
meaningful part of the Householder reflectors.
  If $side = 'L', m \geq l \geq 0$, if $side = 'R', n \geq l \geq 0$.

$a$
    (local)
  Pointer into the local memory to an array of size $lld_a*LOCc(ja+m-1)$ if
$side='L'$, and $lld_a*LOCc(ja+n-1)$ if $side='R'$, where $lld_a \geq$
MAX(1,LOCr($ia+k-1$));
  On entry, the $i$-th row must contain the vector which defines the elementary
reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by $p?tzrzf$ in the $k$
rows of its distributed matrix argument $A(ia:ia+k-1,ja:*)$.
  $A(ia:ia+k-1,ja:*)$ is modified by the routine but restored on exit.

$ia$
    (global)
  The row index in the global array $a$ indicating the first row of sub$(A)$.

$ja$
    (global)
  The column index in the global array $a$ indicating the first column of
sub$(A)$.

$desca$
    (global and local)
  Array of size $dlen_-$.
  The array descriptor for the distributed matrix $A$.

$tau$
    (local)
  Array, size LOCc($ia+k-1$).
  This array contains the scalar factors $tau(i)$ of the elementary reflectors
$H(i)$ as returned by $p?tzrzf$. $tau$ is tied to the distributed matrix $A$.

$c$
    (local)
Pointer into the local memory to an array of size $lld_c*LOCc(jc+n-1)$.

On entry, the local pieces of the distributed matrix sub(C).

$ic$ (global)

The row index in the global array $c$ indicating the first row of sub(C).

$jc$ (global)

The column index in the global array $c$ indicating the first column of sub(C).

$descC$ (global and local)

Array of size $dlen_-$.

The array descriptor for the distributed matrix $C$.

$work$ (local)

Array, size ($lwork$)

On exit, $work(1)$ returns the minimal and optimal $lwork$.

$lwork$ (local or global)

The size of the array $work$.

$lwork$ is local input and must be at least

If $side = 'L'$, $lwork >= MpC0 + MAX( MAX( 1, NqC0 ), numroc( numroc( m +IROFFC, mb_a, 0, 0, NPROW ), mb_a, 0, 0, LCMP ) );$

if $side = 'R'$, $lwork >= NqC0 + MAX( 1, MpC0 );$

where LCMP = LCM / NPROW with LCM = ICLM( NPROW, NPCOL ),

IROFFC = MOD( ic-1, MB_C ), ICOFFC = MOD( jc-1, nb_c ),

ICROW = indxg2p( ic, MB_C, MYROW, rsrc_c, NPROW ),

ICCOL = indxg2p( jc, nb_c, MYCOL, csrc_c, NPCOL ),

$MpC0 = numroc( m+IROFFC, MB_C, MYROW, ICROW, NPROW )$,

$NqC0 = numroc( n+ICOFFC, nb_c, MYCOL, ICCOL, NPCOL )$,

$lcm$, $indxg2p$, and $numroc$ are ScaLAPACK tool functions;

$MYROW$, $MYCOL$, $NPROW$ and $NPCOL$ can be determined by calling the subroutine $blacs_gridinfo$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the routine only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by $pxerbla$.

**Output Parameters**

$c$

On exit, sub(C) is overwritten by $Q*sub(C)$ or $Q'*sub(C)$ or sub(C)*$Q'$ or sub(C)*$Q$.

$work$ (local)

Array, size ($lwork$)
On exit, work[0] returns the minimal and optimal lwork.

info
   = 0: successful exit
   < 0: If the i-th argument is an array and the j-th entry had an illegal value, then info = -(i*100+j), if the i-th argument is a scalar and had an illegal value, then info = -i.

Application Notes
Alignment requirements
The distributed submatrices A(ia:*, ja:*) and C(ic:ic+m-1, jc:jc+n-1) must verify some alignment properties, namely the following expressions should be true:
If side = 'L', ( nb_a = MB_C and ICOFFA = IROFFC )
If side = 'R', ( nb_a = nb_c and ICOFFA = ICOFFC and IACOL = ICCOL )

pormrq
Multiplies a general matrix by the orthogonal matrix Q of the RQ factorization formed by pgerqf.

Syntax
void psormrq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);
void pdormrq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The pormrq function overwrites the general real m-by-n distributed matrix sub (C) = C(ic:ic+m-1, jc:jc+n-1) with

side = 'L':
trans = 'N': Q*sub(C)  side = 'R'
trans = 'T': Q^T*sub(C)

side = 'R':
trans = 'N': sub(C)*Q^T
trans = 'T': sub(C)*Q

where Q is a real orthogonal distributed matrix defined as the product of k elementary reflectors
Q = H(1) H(2)... H(k)

as returned by pgerqf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters
side
   (global)
   = 'L': Q or Q^T is applied from the left.
   = 'R': Q or Q^T is applied from the right.
trans (global) = 'N', no transpose, Q is applied.
= 'T', transpose, QT is applied.

m (global) The number of rows in the distributed matrix sub(C) (m ≥ 0).

n (global) The number of columns in the distributed matrix sub(C) (n ≥ 0).

k (global) The number of elementary reflectors whose product defines the
        matrix Q. Constraints:
        If side = 'L', m ≥ k ≥ 0
        If side = 'R', n ≥ k ≥ 0.

a (local) Pointer into the local memory to an array of size lld_a*LOCc(ja+m-1) if
        side = 'L', and lld_a*LOCc(ja+n-1) if side = 'R'.
        The i-th row of the matrix stored in a must contain the vector that defines
        the elementary reflector H(i), ia ≤ i ≤ ia+k-1, as returned by p?gerqf in the
        k rows of its distributed matrix argument A(ia:ia+k-1, ja:ja). A(ia:ia
        +k-1, ja:* ) is modified by the function but restored on exit.

ia, ja (global) The row and column indices in the global matrix A indicating the
        first row and the first column of the submatrix A, respectively.

desca (global and local) array of size dlen_. The array descriptor for the
        distributed matrix A.

tau (local) Array of size LOCc(ja+k-1).
        Contains the scalar factor tau[i] of elementary reflectors H(i+1) as
        returned by p?gerqf (0 ≤ i < LOCc(ja+k-1)). tau is tied to the distributed
        matrix A.

c (local) Pointer into the local memory to an array of local size lld_c*LOCc(jc+n-1).
        Contains the local pieces of the distributed matrix sub(C) to be factored.

ic, jc (global) The row and column indices in the global matrix C indicating the
        first row and the first column of the matrix sub(C), respectively.

desccc (global and local) array of size dlen_. The array descriptor for the
        distributed matrix C.

work (local) Workspace array of size of lwork.
lwork (local or global) size of work, must be at least:
        If side = 'L',
        lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0 +
        numroc(numroc(n+iroffc, mb_a, 0, 0, NPROW), mb_a, 0, 0,
        lcmp), nqc0))*mb_a) + mb_a*mb_a

4 ScaLAPACK Routines
else if side = 'R',
    lwork ≥ max((mb_a*(mb_a-1)) / 2, (mpc0 + nqc0)*mb_a + mb_a*mb_a)
end if

where

lcmp = lcm/NPROW with lcm = ilcm (NPROW, NPCOL),
iroffa = mod(ia-1, mb_a),
icoffa = mod(ja-1, nb_a),
iacol = indxg2p(ja, nb_a, MYCOL, csnc_a, NPCOL),
mqa0 = numroc(n+icoffa, nb_a, MYCOL, iacol, NPCOL),
irrofc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsnc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csnc_c, NPCOL),
mpc0 = numroc(m+irrofc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

**NOTE**

mod(x, y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
Overwritten by the product Q* sub(C), or Q'*sub (C), or sub(C)* Q', or sub(C)* Q

work[0]
On exit work[0] contains the minimum value of lwork required for optimum performance.

info
(global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
p?unmrq
Multiplies a general matrix by the unitary matrix Q of
the RQ factorization formed by p?gerqf.

Syntax
void pcunmrq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau,
MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work,
MKL_INT *lwork, MKL_INT *info);
void pzunmrq (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k,
MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau,
MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work,
MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
This function overwrites the general complex m-by-n distributed matrix sub (C) = C(ic:ic+m-1,jc:jc+n-1)
with

\[
side = 'L' \quad \text{or} \quad side = 'R'
\]
\[
trans = 'N': \quad Q\cdot\text{sub}(C) \quad \text{or} \quad \text{sub}(C)\cdot Q
\]
\[
trans = 'C': \quad Q^H\cdot\text{sub}(C) \quad \text{or} \quad \text{sub}(C)\cdot Q^H
\]

where Q is a complex unitary distributed matrix defined as the product of k elementary reflectors
Q = H(1)\cdot H(2)\cdot...\cdot H(k)

as returned by p?gerqf. Q is of order m if side = 'L' and of order n if side = 'R'.

Input Parameters

side 
= 'L': Q or Q^H is applied from the left.
= 'R': Q or Q^H is applied from the right.

trans 
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, Q^H is applied.

m 
(global) The number of rows in the distributed matrix sub(C), (m≥0).

n 
(global) The number of columns in the distributed matrix sub(C), (n≥0).

k 
(global) The number of elementary reflectors whose product defines the
matrix Q. Constraints:
If side = 'L', m≥k≥0
If side = 'R', n≥k≥0.

a 
(local)
Pointer into the local memory to an array of size \( lld_a \ast \text{LOCc}(ja+m-1) \) if \( \text{side} = 'L' \), and \( lld_a \ast \text{LOCc}(ja+n-1) \) if \( \text{side} = 'R' \). The \( i \)-th row of the matrix stored in \( A \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia \leq i \leq ia+k-1 \), as returned by \( \text{p?gerqf} \) in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1, ja:) \). \( A(ia:ia+k-1, ja:) \) is modified by the function but restored on exit.

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \tau \) (local) Array of size \( \text{LOCc}(ja+k-1) \).

Contains the scalar factor \( \tau[i] \) of elementary reflectors \( H(i+1) \) as returned by \( \text{p?gerqf} \) \((0 \leq i < \text{LOCc}(ja+k-1)) \). \( \tau \) is tied to the distributed matrix \( A \).

\( c \) (local) Pointer into the local memory to an array of local size \( lld_c \ast \text{LOCc}(jc+n-1) \).

Contains the local pieces of the distributed matrix sub\((C)\) to be factored.

\( ic, jc \) (global) The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

\( \text{descc} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( C \).

\( \text{work} \) (local) Workspace array of size of \( lwork \).

\( lwork \) (local or global) size of \( \text{work} \), must be at least:

If \( \text{side} = 'L', \)

\[
lwork \geq \max((mb_a \ast (mb_a-1))/2, (mpc0 + \text{max(mqa0 + numroc(numroc(n+iroffc, mb_a, 0, 0, NPCOL), mb_a, 0, 0, lcmp), nqc0))} \ast mb_a \) + mb_a \ast mb_a)
\]

else if \( \text{side} = 'R', \)

\[
lwork \geq \max((mb_a \ast (mb_a-1))/2, (mpc0 + nqc0) \ast mb_a \) + mb_a \ast mb_a
\]

end if

where

\[
lcmp = lcm/NPROW \ \text{with} \ lcm = \text{lcm(NPROW, NPCOL)},
\]

\( iroffsa = \text{mod}(ia-1, mb_a) \),

\( icoffsa = \text{mod}(ja-1, nb_a) \),

\( iacol = \text{indx2p}(ja, nb_a, MYCOL, csrc_a, NPCOL), \)

\( mqa0 = \text{numroc}(m+icoffa, nb_a, MYCOL, iacol, NPCOL), \)

\( iroffc = \text{mod}(ic-1, mb_c) \),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrcc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrcc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

**NOTE**
mod(x, y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

**Output Parameters**

**c**
Overwritten by the product Q* sub(C) or Q'* sub(C), or sub(C)* Q', or sub(C)* Q

**work[0]**
On exit work[0] contains the minimum value of lwork required for optimum performance.

**info**
(global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

**See Also**
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?tzrzf**
*Reduces the upper trapezoidal matrix A to upper triangular form.*

**Syntax**

void pstzrzf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);

void pdtzrzf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);

void pctzrzf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pztzrzf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?tzrzf function reduces the m-by-n (m ≤ n) real/complex upper trapezoidal matrix sub(A)= A(ia:ia+m-1, ja:ja+n-1) to upper triangular form by means of orthogonal/unitary transformations. The upper trapezoidal matrix A is factored as
A = (R 0)*Z,
where Z is an n-by-n orthogonal/unitary matrix and R is an m-by-m upper triangular matrix.

Input Parameters

m
(global) The number of rows in the matrix sub(A); (m≥0).

n
(global) The number of columns in the matrix sub(A) (n≥0).

a
(local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1).
Contains the local pieces of the m-by-n distributed matrix sub (A) to be factored.

ia, ja
(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca
(global and local) array of size dlen_. The array descriptor for the distributed matrix A.

work
(local)
Workspace array of size of lwork.

lwork
(local or global) size of work, must be at least
lwork≥mb_a*(mp0+nq0+mb_a), where

\[ iroff = \text{mod}(ia-1, mb_a), \]
\[ icoff = \text{mod}(ja-1, nb_a), \]
\[ iarow = \text{indxg2p}(ia, mb_a, MYROW, rsrc_a, NPROW), \]
\[ iacol = \text{indxg2p}(ja, nb_a, MYCOL, csrc_a, NPCOL), \]
\[ mp0 = \text{numroc}(m+iroff, mb_a, MYROW, iarow, NPROW), \]
\[ nq0 = \text{numroc}(n+icoff, nb_a, MYCOL, iacol, NPCOL) \]

NOTE
mod(x,y) is the integer remainder of x/y.

indxg2p and numroc are ScalAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.
If \texttt{lwork} = -1, then \texttt{lwork} is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \texttt{pxerbla}.

**Output Parameters**

\texttt{a}  
On exit, the leading \(m\)-by-\(m\) upper triangular part of sub(\(A\)) contains the upper triangular matrix \(R\), and elements \(m+1\) to \(n\) of the first \(m\) rows of sub (\(A\)), with the array \(\texttt{tau}\), represent the orthogonal/unitary matrix \(Z\) as a product of \(m\) elementary reflectors.

\texttt{work[0]}  
On exit \texttt{work[0]} contains the minimum value of \texttt{lwork} required for optimum performance.

\texttt{tau}  
(Array of size \texttt{LOCr}(\texttt{ia}+\texttt{m}-1).  
Contains the scalar factor of elementary reflectors. \texttt{tau} is tied to the distributed matrix \(A\).

\texttt{info}  
(global)  
= 0: the execution is successful.  
< 0:if the \(i\)-th argument is an array and the \(j\)-th entry, indexed \(j-1\), had an illegal value, then \texttt{info} = -(\(i\)*100+\(j\)); if the \(i\)-th argument is a scalar and had an illegal value, then \texttt{info} = -\(i\).

**Application Notes**

The factorization is obtained by the Householder’s method. The \(k\)-th transformation matrix, \(Z(k)\), which is or whose conjugate transpose is used to introduce zeros into the \((m - k +1)\)-th row of sub(\(A\)), is given in the form

\[
Z(k) = \begin{bmatrix}
  i & 0 \\
  0 & T(k)
\end{bmatrix}
\]

where

\[
T(k) = i - \texttt{tau} \cdot u(k) \cdot u(k)',
\]

\[
u(k) = \begin{bmatrix}
  1 \\
  0 \\
  z(k)
\end{bmatrix}
\]

\texttt{tau} is a scalar and \(Z(k)\) is an \((n - m)\) element vector. \texttt{tau} and \(Z(k)\) are chosen to annihilate the elements of the \(k\)-th row of sub(\(A\)). The scalar \texttt{tau} is returned in the \(k\)-th element of \texttt{tau}, indexed \(k-1\), and the vector \(u(k)\) in the \(k\)-th row of sub(\(A\)), such that the elements of \(Z(k)\) are in \(a(k, m + 1), \ldots, a(k, n)\). The elements of \(R\) are returned in the upper triangular part of sub(\(A\)). \(Z\) is given by

\[
Z = Z(1) \times Z(2) \times \ldots \times Z(m).
\]
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ormrz**
*Multiplies a general matrix by the orthogonal matrix from a reduction to upper triangular form formed by p?tzrzf.*

**Syntax**
```c
void psormrz (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *i, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pdormrz (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *i, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**
- mkl_scalapack.h

**Description**
This function overwrites the general real $m$-by-$n$ distributed matrix $\text{sub}(C) = C(\text{ic}:\text{ic}+m-1,\text{jc}:\text{jc}+n-1)$ with

- $\text{side} = \text{'L'}$: $Q$ or $Q^T$ is applied from the left.
- $\text{side} = \text{'R'}$: $Q$ or $Q^T$ is applied from the right.

- $\text{trans} = \text{'N'}$: no transpose, $Q$ is applied.
- $\text{trans} = \text{'T'}$: transpose, $Q^T$ is applied.

where $Q$ is a real orthogonal distributed matrix defined as the product of $k$ elementary reflectors $Q = H(1) H(2) \ldots H(k)$ as returned by p?tzrzf. $Q$ is of order $m$ if $\text{side} = \text{'L'}$ and of order $n$ if $\text{side} = \text{'R'}$.

**Input Parameters**
- **side** (global)
  - $\text{side} = \text{'L'}$: $Q$ or $Q^T$ is applied from the left.
  - $\text{side} = \text{'R'}$: $Q$ or $Q^T$ is applied from the right.
- **trans** (global)
  - $\text{trans} = \text{'N'}$: no transpose, $Q$ is applied.
  - $\text{trans} = \text{'T'}$: transpose, $Q^T$ is applied.
- **m** (global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).
- **n** (global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).
- **k** (global) The number of elementary reflectors whose product defines the matrix $Q$. Constraints:
  - If $\text{side} = \text{'L'}$, $m \geq k \geq 0$
If \( \text{side} = 'R' \), \( n \geq k \geq 0 \).

The columns of the distributed matrix sub(\( A \)) containing the meaningful part of the Householder reflectors.

If \( \text{side} = 'L' \), \( m \geq 1 \geq 0 \)

If \( \text{side} = 'R' \), \( n \geq 1 \geq 0 \).

\( a \) (local)

Pointer into the local memory to an array of size \( \text{lld}_a \* \text{LOCc}(ja+m-1) \) if \( \text{side} = 'L' \), and \( \text{lld}_a \* \text{LOCc}(ja+n-1) \) if \( \text{side} = 'R' \), where \( \text{lld}_a \geq \max(1, \text{LOCr}(ia+k-1)) \).

The \( i \)-th row of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia \leq i \leq ia+k-1 \), as returned by \text{p?tzrzf} in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1, ja:*) \). \( A(ia:ia+k-1, ja:*) \) is modified by the function but restored on exit.

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( \text{desca} \) (global and local) array of size \( \text{dlen}_a \). The array descriptor for the distributed matrix \( A \).

\( \tau \) (local)

Array of size \( \text{LOCc}(ia+k-1) \).

Contains the scalar factor \( \tau[i] \) of elementary reflectors \( H(i+1) \) as returned by \text{p?tzrzf} \( (0 \leq i < \text{LOCc}(ia+k-1)) \). \( \tau \) is tied to the distributed matrix \( A \).

\( c \) (local)

Pointer into the local memory to an array of local size \( \text{lld}_c \* \text{LOCc}(jc+n-1) \).

Contains the local pieces of the distributed matrix sub(\( C \)) to be factored.

\( ic, jc \) (global) The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

\( \text{descC} \) (global and local) array of size \( \text{dlen}_c \). The array descriptor for the distributed matrix \( C \).

\( \text{work} \) (local)

Workspace array of size of \( \text{lwork} \).

\( \text{lwork} \) (local or global) size of \( \text{work} \), must be at least:

If \( \text{side} = 'L' \),

\[ \text{lwork} \geq \max( \frac{(\text{mb}_a \* (\text{mb}_a-1))}{2}, (\text{mpc}_0 + \max(\text{mqa}_0 + \text{numroc}(\text{numroc}(n+ioffc, \text{mb}_a, 0, 0, \text{NPROW}), \text{mb}_a, 0, 0, \text{lcmp}), \text{nqc}_0) \* \text{mb}_a) + \text{mb}_a \* \text{mb}_a \) \]

else if \( \text{side} = 'R' \),

\[ \text{lwork} \geq \max( \frac{(\text{mb}_a \* (\text{mb}_a-1))}{2}, (\text{mpc}_0 + \text{nqc}_0) \* \text{mb}_a) + \text{mb}_a \* \text{mb}_a \) \]
end if

where

\[ lcmp = \text{lcm}/\text{NPROW} \text{ with } \text{lcm} = \text{ilcm} (\text{NPROW}, \text{NPCOL}), \]
\[ \text{iroffa} = \text{mod}(\text{ia}-1, \text{mb}_a), \text{icoffa} = \text{mod}(\text{ja}-1, \text{nb}_a), \]
\[ \text{iacol} = \text{indxg2p}(\text{ja}, \text{nb}_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}), \]
\[ \text{mqa0} = \text{numroc}(\text{n}+\text{icoffa}, \text{nb}_a, \text{MYCOL}, \text{iacol}, \text{NPCOL}), \]
\[ \text{iroffc} = \text{mod}(\text{ic}-1, \text{mb}_c), \]
\[ \text{icoffc} = \text{mod}(\text{jc}-1, \text{nb}_c), \]
\[ \text{iacon} = \text{indxg2p}(\text{ja}, \text{nb}_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}), \]
\[ \text{mpc0} = \text{numroc}(\text{m}+\text{iroffc}, \text{mb}_c, \text{MYROW}, \text{icrow}, \text{NPROW}), \]
\[ \text{nqc0} = \text{numroc}(\text{n}+\text{icoffc}, \text{nb}_c, \text{MYCOL}, \text{iccol}, \text{NPCOL}), \]

**NOTE**

\[ \text{mod}(x,y) \text{ is the integer remainder of } x/y. \]

\text{ilcm}, \text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; } \text{MYROW}, \text{MYCOL}, \text{NPROW} \text{ and } \text{NPCOL} \text{ can be determined by calling the function } \text{blacs_gridinfo}. \]

If \text{lwork} = -1, then \text{lwork} is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

**Output Parameters**

\text{c}

Overwritten by the product \( Q*\text{sub}(C) \), or \( Q^\ast*\text{sub}(C) \), or \( \text{sub}(C)*Q^\ast \), or \( \text{sub}(C)*Q \)

\text{work[0]}

On exit \text{work[0]} contains the minimum value of \text{lwork} required for optimum performance.

\text{info}

(global)

\[ = 0: \text{the execution is successful.} \]
\[ < 0: \text{if the } i\text{-th argument is an array and the } j\text{-th entry, indexed } j - 1, \text{ had an illegal value, then } \text{info} = -(i*100+j); \text{ if the } i\text{-th argument is a scalar and had an illegal value, then } \text{info} = -i. \]

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

\text{p?unmrz}

Multiplies a general matrix by the unitary transformation matrix from a reduction to upper triangular form determined by \text{p?tzrzf}. 

Syntax

```c
void pcunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT *
1 , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau ,
MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex8 *work ,
MKL_INT *lwork , MKL_INT *info );
```

```c
void pzunmrz (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_INT *
1 , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *
tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descc , MKL_Complex16 *
work , MKL_INT *lwork , MKL_INT *info );
```

Include Files

- mkl_scalapack.h

Description

This function overwrites the general complex m-by-n distributed matrix

\[
\text{sub}(C) = C(i\text{c}:i\text{c}+m-1,j\text{c}:j\text{c}+n-1)
\]

with

\[
\begin{align*}
\text{side} = 'L' & \quad \text{side} = 'R' \\
\text{trans} = 'N' : & \quad Q*\text{sub}(C) & \quad \text{sub}(C)*Q \\
\text{trans} = 'C' : & \quad Q^H*\text{sub}(C) & \quad \text{sub}(C)*Q^H
\end{align*}
\]

where \(Q\) is a complex unitary distributed matrix defined as the product of \(k\) elementary reflectors

\(Q = H(1)' H(2)'... H(k)\)

as returned by pctzrzf/pztzrzf. \(Q\) is of order \(m\) if \text{side} = 'L' and of order \(n\) if \text{side} = 'R'.

Input Parameters

- \text{side} (global)
  - 'L': Q or \(Q^H\) is applied from the left.
  - 'R': Q or \(Q^H\) is applied from the right.

- \text{trans} (global)
  - 'N', no transpose, Q is applied.
  - 'C', conjugate transpose, \(Q^H\) is applied.

- \text{m} (global) The number of rows in the distributed matrix \(\text{sub}(C)\), \((m \geq 0)\).

- \text{n} (global) The number of columns in the distributed matrix \(\text{sub}(C)\), \((n \geq 0)\).

- \text{k} (global) The number of elementary reflectors whose product defines the matrix \(Q\). Constraints:
  - If \text{side} = 'L', \(m \geq k \geq 0\)
  - If \text{side} = 'R', \(n \geq k \geq 0\).

- \text{l} (global) The columns of the distributed matrix \(\text{sub}(A)\) containing the meaningful part of the Householder reflectors.
  - If \text{side} = 'L', \(m \geq l \geq 0\)
  - If \text{side} = 'R', \(n \geq l \geq 0\).
Pointer into the local memory to an array of size \( lld_a*\text{LOCc}(ja+m-1) \) if \( \text{side} = 'L' \), and \( lld_a*\text{LOCc}(ja+n-1) \) if \( \text{side} = 'R' \), where \( lld_a \geq \text{max}(1, \text{LOCr}(ja+k-1)) \). The \( i \)-th row of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia \leq i \leq ia+k-1 \), as returned by \( \text{p?gerqf} \) in the \( k \) rows of its distributed matrix argument \( A(ia:ia+k-1, ja:*) \). \( A(ia:ia+k-1, ja:*) \) is modified by the function but restored on exit.

**ia, ja**

The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

**desca**

Array of size \( \text{LOCc}(ia+k-1) \).

Contains the scalar factor \( \tau[i] \) of elementary reflectors \( H(i+1) \) as returned by \( \text{p?gerqf} \) \( (0 \leq i < \text{LOCc}(ia+k-1)) \). \( \tau \) is tied to the distributed matrix \( A \).

**c**

Pointer into the local memory to an array of local size \( lld_c*\text{LOCc}(jc+n-1) \).

Contains the local pieces of the distributed matrix sub(\( C \)) to be factored.

**ic, jc**

The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

**dexec**

Array of size \( \text{dlen}_c \). The array descriptor for the distributed matrix \( C \).

**work**

Workspace array of size \( lwork \).

**lwork**

Size of \( \text{work} \), must be at least:

If \( \text{side} = 'L' \),

\[
lwork \geq \max\left(\frac{(mb_a*(mb_a-1))}{2}, \frac{(mpc0+\text{max}(mqa0+\text{numroc}(\text{numroc}(n+iwoffc, mb_a, 0, 0, \text{NPROW}), mb_a, 0, 0, lcmp), nqc0))*mb_a}{mb_a} + mb_a*mb_a\right)
\]

else if \( \text{side} = 'R' \),

\[
lwork \geq \max\left(\frac{(mb_a*(mb_a-1))}{2}, \frac{(mpc0+nqc0)*mb_a}{mb_a} + mb_a*mb_a\right)
\]

end if

where

\[
lcmp = \text{lcm}/\text{NPROW} \text{ with lcm = ilcm(\text{NPROW}, \text{NPCOL})},
\]

\[
iwoffc = \text{mod}(ia-1, mb_a),
\]

\[
\text{icoffa} = \text{mod}(ja-1, nb_a),
\]

\[
\text{iacol} = \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}),
\]

\[
mqa0 = \text{numroc}(m+icoffa, nb_a, \text{MYCOL}, \text{iacol}, \text{NPCOL}),
\]

\[
\text{mpc0} = \text{numroc}(n+iwoffc, mb_a, 0, 0, \text{NPROW}),
\]

\[
\text{nqc0} = \text{numroc}(n+iwoffc, mb_a, 0, 0, lcmp).
\]
iroffc = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, MYROW, rsrc_c, NPROW),
iccol = indxg2p(jc, nb_c, MYCOL, csrc_c, NPCOL),
mpc0 = numroc(m+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(n+icoffc, nb_c, MYCOL, iccol, NPCOL),

**NOTE**

mod(x, y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

### Output Parameters

**c**

Overwritten by the product $Q^* \text{sub}(C)$, or $Q^* \text{sub}(C)$, or $\text{sub}(C)*Q'$, or $\text{sub}(C)*Q$

**work[0]**

On exit work[0] contains the minimum value of lwork required for optimum performance.

**info**

(global)

= 0: the execution is successful.

< 0: if the i-th argument is an array and the j-th entry, indexed $j-1$, had an illegal value, then info = -$i*100+j$; if the i-th argument is a scalar and had an illegal value, then info = -$i$.

### See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ggqr**

Computes the generalized QR factorization.

### Syntax

void psggqrf (MKL_INT *n, MKL_INT *m, MKL_INT *p, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tauA, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, float *tauB, float *work, MKL_INT *lwork, MKL_INT *info);

void pdggqr (MKL_INT *n, MKL_INT *m, MKL_INT *p, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tauA, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, double *tauB, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgqr (MKL_INT *n, MKL_INT *m, MKL_INT *p, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex8 *taub, MKL_Complex8 *work, MKL_INT *iwork, MKL_INT *info);
void pzgqr (MKL_INT *n, MKL_INT *m, MKL_INT *p, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex16 *taub, MKL_Complex16 *work, MKL_INT *iwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The p?gqr function forms the generalized QR factorization of an \(n\)-by-\(m\) matrix

\[
\text{sub}(A) = A(ia:ia+n-1, ja:ja+m-1)
\]

and an \(n\)-by-\(p\) matrix

\[
\text{sub}(B) = B(ib:ib+n-1, jb:jb+p-1)
\]

as

\[
\text{sub}(A) = Q*R, \text{sub}(B) = Q*T*Z,
\]

where \(Q\) is an \(n\)-by-\(n\) orthogonal/unitary matrix, \(Z\) is a \(p\)-by-\(p\) orthogonal/unitary matrix, and \(R\) and \(T\) assume one of the forms:

If \(n \geq m\)

\[
R = \begin{pmatrix} R_{11} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix}
\]

where \(R_{11}\) is upper triangular, and

\[
T = \begin{pmatrix} \mathbf{0} & T_{12} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} n, \text{ if } n \leq p,
\]

\[
T = \begin{pmatrix} T_{11} & \mathbf{0} \\ T_{21} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{pmatrix} n - p, \text{ if } n > p,
\]

where \(T_{12}\) or \(T_{21}\) is an upper triangular matrix.
In particular, if sub\((B)\) is square and nonsingular, the GQR factorization of sub\((A)\) and sub\((B)\) implicitly gives the QR factorization of \(\text{inv}(\text{sub}(B)) \ast \text{sub}(A)\):

\[
\text{inv}(\text{sub}(B)) \ast \text{sub}(A) = Z^H \ast (\text{inv}(T) \ast R)
\]

**Input Parameters**

\(n\)

(global) The number of rows in the distributed matrices sub\((A)\) and sub\((B)\) \((n \geq 0)\).

\(m\)

(global) The number of columns in the distributed matrix sub\((A)\) \((m \geq 0)\).

\(p\)

The number of columns in the distributed matrix sub\((B)\) \((p \geq 0)\).

\(a\)

(local)

Pointer into the local memory to an array of size \(lld_a \ast \text{LOCc}(ja+m-1)\).

Contains the local pieces of the \(n\)-by-\(m\) matrix sub\((A)\) to be factored.

\(ia, ja\)

(global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.

\(desca\)

(global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(A\).

\(b\)

(local)

Pointer into the local memory to an array of size \(lld_b \ast \text{LOCc}(jb+p-1)\).

Contains the local pieces of the \(n\)-by-\(p\) matrix sub\((B)\) to be factored.

\(ib, jb\)

(global) The row and column indices in the global matrix \(B\) indicating the first row and the first column of the submatrix \(B\), respectively.

\(descb\)

(global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(B\).

\(work\)

(local)

Workspace array of size of \(lwork\).

\(lwork\)

(local or global) Sze of \(work\), must be at least

\[
\text{lwork} \geq \max\left(nb_a \ast (npa0+ma0+nb_a), \max((nb_a \ast (nb_a-1))/2, (pqb0+npb0) \ast nb_a) + nb_a \ast nb_a, mb_b \ast (npb0+pqb0+mb_b)\right),
\]

where

\[
\begin{align*}
\text{iroffa} &= \text{mod}(ia-1, mb_A), \\
\text{icoffa} &= \text{mod}(ja-1, nb_a), \\
\text{iarow} &= \text{indxg2p}(ia, mb_a, \text{MYROW}, \text{rsrc}_a, \text{NPROW}), \\
\text{iacol} &= \text{indxg2p}(ja, nb_a, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}), \\
\text{npa0} &= \text{numroc} (n+\text{iroffa}, mb_a, \text{MYROW}, \text{iarow}, \text{NPROW}), \\
\text{ma0} &= \text{numroc} (m+\text{icoffa}, nb_a, \text{MYCOL}, \text{iacol}, \text{NPCOL}) \\
\text{iroffb} &= \text{mod}(ib-1, mb_b), \\
\text{icoffb} &= \text{mod}(jb-1, nb_b), \\
\text{ibrow} &= \text{indxg2p}(ib, mb_b, \text{MYROW}, \text{rsrc}_b, \text{NPROW}),
\end{align*}
\]
ibcol = indxg2p(jb, nb_b, MYCOL, csrc_b, NPCOL),
npb0 = numroc (n+iroffa, mb_b, MYROW, Ibrow, NPROW),
pqb0 = numroc(m+icoffb, nb_b, MYCOL, ibcol, NPCOL)

NOTE
\mod(x,y) \text{ is the integer remainder of } x/y.

and numroc, indxg2p are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pzerbla.

Output Parameters

\textbf{a}

On exit, the elements on and above the diagonal of sub (A) contain the min(n, m)-by-m upper trapezoidal matrix \( R \) (\( R \) is upper triangular if \( n \geq m \)); the elements below the diagonal, with the array \( \text{taua} \), represent the orthogonal/unitary matrix \( Q \) as a product of \( \min(n, m) \) elementary reflectors. (See Application Notes below).

\textbf{taua}, \textbf{taub}

Arrays of size \( \text{LOCc}(ja+\min(n,m)-1) \) for \( \text{taua} \) and \( \text{LOCr}(ib+n-1) \) for \( \text{taub} \).

The array \( \text{taua} \) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Q \). \( \text{taua} \) is tied to the distributed matrix \( A \). (See Application Notes below).

The array \( \text{taub} \) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Z \). \( \text{taub} \) is tied to the distributed matrix \( B \). (See Application Notes below).

\textbf{work[0]}

On exit \( \text{work[0]} \) contains the minimum value of \( lwork \) required for optimum performance.

\textbf{info}

(global)

\( = 0: \) the execution is successful.

\( < 0: \) if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors

\[ Q = H(ja)*H(ja+1)*...*H(ja+k-1), \]

where \( k = \min(n,m) \).

Each \( H(i) \) has the form

\[ H(i) = i - \text{taua}^*v*v' \]
where \( \tau_a \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(1:i-1) = 0 \) and \( v(i) = 1 \); \( v(i+1:n) \) is stored on exit in \( A(ia:i+a+n-1, ja:i-1) \), and \( \tau_a \) in \( \tau_a[ja+i-2] \). To form \( Q \) explicitly, use ScaLAPACK function \( \text{p?orgqr/p?unqr} \). To use \( Q \) to update another matrix, use ScaLAPACK function \( \text{p?ormqr/p?unmqr} \).

The matrix \( Z \) is represented as a product of elementary reflectors
\[
Z = H(ib)^*H(ib+1)^*...*H(ib+k-1), \text{ where } k = \min(n,p).
\]

Each \( H(i) \) has the form
\[
H(i) = I - \tau_a^* v' v
\]
where \( \tau_a \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(p-k+i+1:p) = 0 \) and \( v(p-k+i) = 1 \); \( v(1:p-k+i-1) \) is stored on exit in \( B(ib+n-k+i-1, jb:jb+p-k+i-2) \), and \( \tau_a \) in \( \tau_a[ib+n-k+i-2] \). To form \( Z \) explicitly, use ScaLAPACK function \( \text{p?orgqr/p?unqr} \). To use \( Z \) to update another matrix, use ScaLAPACK function \( \text{p?ormqr/p?unmqr} \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ggrqf**

Computes the generalized RQ factorization.

**Syntax**

```c
void psggrqf (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tauaf, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, float *tauab, float *work, MKL_INT *lwork, MKL_INT *info);
void pdggrqf (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tauad, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, double *tauab, double *work, MKL_INT *lwork, MKL_INT *info);
void pcggrqf (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *taua, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex8 *taub, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzggrqf (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *taua, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex16 *taub, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The \( \text{p?ggrqf} \) function forms the generalized RQ factorization of an \( m \)-by-\( n \) matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) and a \( p \)-by-\( n \) matrix \( \text{sub}(B) = B(ib:ib+p-1, jb:jb+n-1) \):

\[
\text{sub}(A) = R^*Q, \quad \text{sub}(B) = Z^*T^*Q,
\]

where \( Q \) is an \( n \)-by-\( n \) orthogonal/unitary matrix, \( Z \) is a \( p \)-by-\( p \) orthogonal/unitary matrix, and \( R \) and \( T \) assume one of the forms:

\[
R = \begin{cases} 
0 & R_{12} \\
R_{12} & \end{cases}, \text{ if } m \leq n,
\]

\[
R_{12} = \begin{pmatrix} R_{11} & R_{12} \\
R_{21} & R_{22} \end{pmatrix}, \text{ if } m > n,
\]

\[
T = \begin{cases} 
I & T_{22} \\
T_{21} & \end{cases}, \text{ if } n \leq p,
\]

\[
T_{21} = \begin{pmatrix} T_{211} & T_{212} \\
T_{11} & T_{12} \end{pmatrix}, \text{ if } n > p.
\]
or
\[
R = \begin{pmatrix}
R_{11} & m - n \\
R_{12} & n
\end{pmatrix}, \quad \text{if } m > n
\]
where \(R_{11}\) or \(R_{21}\) is upper triangular, and
\[
T = \begin{pmatrix}
T_{11} & n \\
0 & p - n
\end{pmatrix}, \quad \text{if } p \geq n
\]
or
\[
T = p \begin{pmatrix}
T_{11} & T_{12} \\
0 & n - p
\end{pmatrix}, \quad \text{if } p < n,
\]
where \(T_{11}\) is upper triangular.

In particular, if \(\text{sub}(B)\) is square and nonsingular, the GRQ factorization of \(\text{sub}(A)\) and \(\text{sub}(B)\) implicitly gives the RQ factorization of \(\text{sub}(A)^* \text{inv}(\text{sub}(B))\):
\[
\text{sub}(A)^* \text{inv}(\text{sub}(B)) = (R^* \text{inv}(T))^* Z^\prime
\]
where \(\text{inv}(\text{sub}(B))\) denotes the inverse of the matrix \(\text{sub}(B)\), and \(Z^\prime\) denotes the transpose (conjugate transpose) of matrix \(Z\).

**Input Parameters**

- \(m\) (global) The number of rows in the distributed matrices \(\text{sub}(A)\) \((m \geq 0)\).
- \(p\) The number of rows in the distributed matrix \(\text{sub}(B)\) \((p \geq 0)\).
- \(n\) (global) The number of columns in the distributed matrices \(\text{sub}(A)\) and \(\text{sub}(B)\) \((n \geq 0)\).
- \(a\) (local)
  Pointer into the local memory to an array of size \(lld_a * \text{LOCc}(ja+n-1)\).
  Contains the local pieces of the \(m\)-by-\(n\) distributed matrix \(\text{sub}(A)\) to be factored.
- \(ia, ja\) (global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.
- \(desca\) (global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(A\).
- \(b\) (local)
  Pointer into the local memory to an array of size \(lld_b * \text{LOCc}(jb+n-1)\).
  Contains the local pieces of the \(p\)-by-\(n\) matrix \(\text{sub}(B)\) to be factored.
(global) The row and column indices in the global matrix $B$ indicating the first row and the first column of the submatrix $B$, respectively.

(descb) array of size dlen_. The array descriptor for the distributed matrix $B$.

(work) Workspace array of size of lwork.

(lwork) Size of work, must be at least 
\[
\text{lwork} \geq \max(\text{mb}_a*\text{mp}_a + \text{nqa}_a + \text{mb}_a, \text{mb}_a*\text{pp}_b + \text{nqb}_b + \text{nb}_b),
\]
where
\[
\text{iroffa} = \text{mod}(\text{ia} - 1, \text{mb}_A),
\]
\[
\text{icoffa} = \text{mod}(\text{ja} - 1, \text{nb}_A),
\]
\[
\text{iarow} = \text{indxg2p}(\text{ia}, \text{mb}_A, \text{MYROW}, \text{rsrc}_A, \text{NPROW}),
\]
\[
\text{iacol} = \text{indxg2p}(\text{ja}, \text{nb}_A, \text{MYCOL}, \text{csrc}_A, \text{NPCOL}),
\]
\[
\text{mp}_a = \text{numroc}(\text{m} + \text{iroffa}, \text{mb}_A, \text{MYROW}, \text{iarow}, \text{NPROW}),
\]
\[
\text{nqa}_a = \text{numroc}(\text{m} + \text{icoffa}, \text{nb}_A, \text{MYCOL}, \text{iacol}, \text{NPCOL})
\]
\[
\text{iroffb} = \text{mod}(\text{ib} - 1, \text{mb}_B),
\]
\[
\text{icoffb} = \text{mod}(\text{jb} - 1, \text{nb}_B),
\]
\[
\text{ibrow} = \text{indxg2p}(\text{ib}, \text{mb}_B, \text{MYROW}, \text{rsrc}_B, \text{NPROW}),
\]
\[
\text{ibcol} = \text{indxg2p}(\text{jb}, \text{nb}_B, \text{MYCOL}, \text{csrc}_B, \text{NPCOL}),
\]
\[
\text{pp}_b = \text{numroc}(\text{p} + \text{iroffb}, \text{mb}_B, \text{MYROW}, \text{ibrow}, \text{NPROW}),
\]
\[
\text{nqb}_b = \text{numroc}(\text{n} + \text{icoffb}, \text{nb}_B, \text{MYCOL}, \text{ibcol}, \text{NPCOL})
\]

NOTE 
\[
\text{mod}(x, y) \text{ is the integer remainder of } x/y.
\]

and \text{numroc}, \text{indxg2p} are ScaLAPACK tool functions; \text{MYROW}, \text{MYCOL}, \text{NPROW} and \text{NPCOL} can be determined by calling the function \text{blacs_gridinfo}.

If \text{lwork} = -1, then \text{lwork} is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

(a) On exit, if $m \leq n$, the upper triangle of $A(ia:ia+m-1, ja+n-m:ja+n-1)$ contains the $m$-by-$m$ upper triangular matrix $R$; if $m \geq n$, the elements on and above the $(m-n)$-th subdiagonal contain the $m$-by-$n$ upper trapezoidal matrix $R$; the remaining elements, with the array \text{taua}, represent the orthogonal/unitary matrix $Q$ as a product of $\min(n,m)$ elementary reflectors (see Application Notes below).

(taua, taub) (local)
Arrays of size \( \text{LOCr}(ia+m-1) \) for \( \text{taua} \) and \( \text{LOCc}(jb+\min(p,n)-1) \) for \( \text{taub} \).

The array \( \text{taua} \) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Q \). \( \text{taua} \) is tied to the distributed matrix \( A \). (See Application Notes below).

The array \( \text{taub} \) contains the scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Z \). \( \text{taub} \) is tied to the distributed matrix \( B \). (See Application Notes below).

\( \text{work}[0] \)

On exit \( \text{work}[0] \) contains the minimum value of \( l\text{work} \) required for optimum performance.

\( \text{info} \)

\( \text{(global)} \)

\( = 0 \): the execution is successful.

\(< 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1 \), had an illegal value, then \( \text{info} = -(i\times100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors

\[ Q = H(ia)H(ia+1)\ldots H(ia+k-1), \]

where \( k = \min(m,n) \).

Each \( H(i) \) has the form

\[ H(i) = i - \text{taua}v^*v' \]

where \( \text{taua} \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(n-k+i+1:n) = 0 \) and \( v(n-k+i) = 1 \); \( v(1:n-k+i-1) \) is stored on exit in \( A(ia+m-k+i-1, ja:ja+n-k+i-2) \), and \( \text{taua} \) in \( \text{taua}[ia+m-k+i-2] \). To form \( Q \) explicitly, use ScaLAPACK function \( \text{p?orgqr/p?ungqr} \). To use \( Q \) to update another matrix, use ScaLAPACK function \( \text{p?ormqr/p?unmqr} \).

The matrix \( Z \) is represented as a product of elementary reflectors

\[ Z = H(jb)H(jb+1)\ldots H(jb+k-1), \]

where \( k = \min(p,n) \).

Each \( H(i) \) has the form

\[ H(i) = i - \text{taub}v^*v' \]

where \( \text{taub} \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(1:i-1) = 0 \) and \( v(i) = 1 \); \( v(i+1:p) \) is stored on exit in \( B(ib+i:ib+p-1,jb+i-1) \), and \( \text{taub} \) in \( \text{taub}[jb+i-2] \). To form \( Z \) explicitly, use ScaLAPACK function \( \text{p?orgqr/p?ungqr} \). To use \( Z \) to update another matrix, use ScaLAPACK function \( \text{p?ormqr/p?unmqr} \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

Symmetric Eigenvalue Problems: ScaLAPACK Computational Routines

To solve a symmetric eigenproblem with ScaLAPACK, you usually need to reduce the matrix to real tridiagonal form \( T \) and then find the eigenvalues and eigenvectors of the tridiagonal matrix \( T \). ScaLAPACK includes routines for reducing the matrix to a tridiagonal form by an orthogonal (or unitary) similarity transformation \( A = QQT^T \) as well as for solving tridiagonal symmetric eigenvalue problems. These routines are listed in Table “Computational Routines for Solving Symmetric Eigenproblems”.
There are different routines for symmetric eigenproblems, depending on whether you need eigenvalues only or eigenvectors as well, and on the algorithm used (either the QTQ algorithm, or bisection followed by inverse iteration).

### Computational Routines for Solving Symmetric Eigenproblems

<table>
<thead>
<tr>
<th>Operation</th>
<th>Dense symmetric/Hermitian matrix</th>
<th>Orthogonal/unitary matrix</th>
<th>Symmetric tridiagonal matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce to tridiagonal form $A = QTQ^H$</td>
<td>$\text{p?sytrd/p?hetrd}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply matrix after reduction</td>
<td></td>
<td>$\text{p?ormtr/p?unmtr}$</td>
<td></td>
</tr>
<tr>
<td>Find all eigenvalues and eigenvectors of a tridiagonal matrix $T$ by a QTQ method</td>
<td></td>
<td></td>
<td>$\text{steqr2}^*$</td>
</tr>
<tr>
<td>Find selected eigenvalues of a tridiagonal matrix $T$ via bisection</td>
<td></td>
<td></td>
<td>$\text{p?stebz}$</td>
</tr>
<tr>
<td>Find selected eigenvectors of a tridiagonal matrix $T$ by inverse iteration</td>
<td></td>
<td></td>
<td>$\text{p?stein}$</td>
</tr>
</tbody>
</table>

* This routine is described as part of auxiliary ScaLAPACK routines.

#### p?syngst

*Reduces a complex Hermitian-definite generalized eigenproblem to standard form.*

### Syntax

```c
void pssyngst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const float* b, const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, float* scale, float* work, const MKL_INT* lwork, MKL_INT* info);

void pdsyngst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const double* b, const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, double* scale, double* work, const MKL_INT* lwork, MKL_INT* info);
```

### Include Files

- mkl_scalapack.h

### Description

p?syngst *reduces a complex Hermitian-definite generalized eigenproblem to standard form.*

p?syngst *performs the same function as p?hegst, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of p?syngst).*

p?syngst *calls p?hegst when uplo='U', hence p?hegst provides improved performance only when uplo='L', ibtype=1.*

p?syngst *also calls p?hegst when insufficient workspace is provided, hence p?syngst provides improved performance only when lwork >= 2 * NPO * NB + NQO * NB + NB * NB.*

In the following sub($A$) denotes $A(ia:ia+n-1,ja:ja+n-1)$ and sub($B$) denotes $B(ib:ib+n-1,jb:jb+n-1)$.

If $ibtype = 1$, the problem is sub($A$)*$x = \lambda$*sub($B$)*$x$, and sub($A$) is overwritten by inv($U^H$)*sub($A$)*$x$ or inv($L$)*sub($A$)*$x$.

If $ibtype = 2$ or $3$, the problem is sub($A$)*sub($B$)*$x = \lambda$*sub($B$)*$x$ or sub($B$)*sub($A$)*$x = \lambda$*sub($A$)*$x$, and sub($A$) is overwritten by $U$*sub($A$)*$U^H$ or $L^H$*sub($A$)*$L$. 

1365
sub( B ) must have been previously factorized as $U^H U$ or $L^*L^H$ by p?potrf.

### Input Parameters

- **ibtype**
  - (global)
  - $= 1$: compute $\text{inv}(U^H)\text{sub}(A)\text{inv}(U)$ or $\text{inv}(L)\text{sub}(A)\text{inv}(L^H)$;
  - $= 2$ or $3$: compute $U\text{sub}(A)U^H$ or $L^*\text{sub}(A)L$.

- **uplo**
  - (global)
  - $= 'U'$: Upper triangle of sub( A ) is stored and sub( B ) is factored as $U^H U$;
  - $= 'L'$: Lower triangle of sub( A ) is stored and sub( B ) is factored as $L^*L^H$.

- **n**
  - (global)
  - The order of the matrices sub( A ) and sub( B ). $n \geq 0$.

- **a**
  - (local)
  - Pointer into the local memory to an array of size $\text{lld}_a\text{LOC}(ja+n-1)$.
  - On entry, this array contains the local pieces of the $n$-by-$n$ Hermitian distributed matrix sub( A ). If $\text{uplo} = 'U'$, the leading $n$-by-$n$ upper triangular part of sub( A ) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If $\text{uplo} = 'L'$, the leading $n$-by-$n$ lower triangular part of sub( A ) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

- **ia**
  - (global)
  - A's global row index, which points to the beginning of the submatrix which is to be operated on.

- **ja**
  - (global)
  - A's global column index, which points to the beginning of the submatrix which is to be operated on.

- **desca**
  - (global and local)
  - Array of size $\text{dien}_\_$. The array descriptor for the distributed matrix A.

- **b**
  - (local)
  - Pointer into the local memory to an array of size $\text{lld}_b\text{LOC}(jb+n-1)$.
  - On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub( B ), as returned by p?potrf.

- **ib**
  - (global)
  - B's global row index, which points to the beginning of the submatrix which is to be operated on.

- **jb**
  - (global)
  - B's global column index, which points to the beginning of the submatrix which is to be operated on.

- **descb**
  - (global and local)
The array descriptor for the distributed matrix $B$.

(work)

Array, size (lwork)

(lwork)

The size of the array work.

lwork is local input and must be at least $lwork \geq \text{MAX}( \text{NB} \times (\text{NP0} + 1), 3 \times \text{NB} )$

When $\text{ibtype} = 1$ and $\text{uplo} = 'L'$, \text{p?syngst} provides improved performance when $lwork \geq 2 \times \text{NP0} \times \text{NB} + \text{NQ0} \times \text{NB} + \text{NB} \times \text{NB}$,

where $\text{NB} = \text{mb}_a = \text{nb}_a$,

$\text{NP0} = \text{numroc}( n, \text{NB}, 0, 0, \text{NPROW} )$,

$\text{NQ0} = \text{numroc}( n, \text{NB}, 0, 0, \text{NPROW} )$,

\text{numroc} is a ScaLAPACK tool functions

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If $lwork = -1$, then lwork is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\text{a}

On exit, if $\text{info} = 0$, the transformed matrix, stored in the same format as sub( $A$ ).

(scale)

Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine. At present, scale is always returned as 1.0, it is returned here to allow for future enhancement.

(work)

Array, size (lwork)

On exit, work[0] returns the minimal and optimal lwork.

(info)

= 0: successful exit

< 0: If the $i$-th argument is an array and the $j$-th entry had an illegal value, then $\text{info} = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.

p?syntrd

\textit{Reduces a real symmetric matrix to symmetric tridiagonal form.}
Syntax

```c
void pssytrd (const char* uplo, const MKL_INT* n, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, float* d, float* e, float* tau, float* work, const MKL_INT* lwork, MKL_INT* info);

void pdsytrd (const char* uplo, const MKL_INT* n, double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, double* d, double* e, double* tau, double* work, const MKL_INT* lwork, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

`p?syntrd` is a prototype version of `p?sytrd` which uses tailored codes (either the serial, `?sytrd`, or the parallel code, `p?syttrd`) when the workspace provided by the user is adequate.

`p?syntrd` reduces a real symmetric matrix `sub(A)` to symmetric tridiagonal form `T` by an orthogonal similarity transformation:

\[ Q' \cdot sub(A) \cdot Q = T, \]

where `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`.

Features

`p?syntrd` is faster than `p?sytrd` on almost all matrices, particularly small ones (i.e. `n < 500 * sqrt(P)`), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on `ia`, `ja`, `MB` or `NB`. At present, `ia`, `ja`, `MB` and `NB` are restricted to those values allowed by `p?hetrd` to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

**uplo**

(global)

Specifies whether the upper or lower triangular part of the symmetric matrix `sub(A)` is stored:

- `='U'`: Upper triangular
- `='L'`: Lower triangular

**n**

(global)

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix `sub(A)`, `n >= 0`.

**a**

(local)

Pointer into the local memory to an array of size `lda*LOCc(ja+n-1)`. On entry, this array contains the local pieces of the symmetric distributed matrix `sub(A)`. If `uplo = 'U'`, the leading `n`-by-`n` upper triangular part of `sub(A)` contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If `uplo = 'L'`, the leading `n`-by-`n` lower triangular part of `sub(A)` contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

**ia**

(global)
The row index in the global array $a$ indicating the first row of sub($A$).

$ja$ (global)

The column index in the global array $a$ indicating the first column of sub($A$).

desca (global and local)

Array of size $dlen_-$.

The array descriptor for the distributed matrix $A$.

work (local)

Array, size ($lwork$)

$lwork$ (local or global)

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq \text{MAX( NB \times ( NP + 1 ), 3 \times NB )}$

For optimal performance, greater workspace is needed, i.e.

$lwork \geq 2 \times ( ANB +1 ) \times ( 4 \times NPS +2 ) + ( NPS + 4 ) \times NPS$

$ANB = \text{pjlaenv}( ICTXT, 3, 'p?sytrd', 'L', 0, 0, 0, 0 )$

$ICTXT = \text{desca}( \text{ctxt}_- )$

$SQNPC = \text{INT( sqrt( REAL( NPROW \times NPCOL ) ) )}$

$numroc$ is a ScaLAPACK tool function.

$\text{pjlaenv}$ is a ScaLAPACK environmental inquiry function.

$NPROW$ and $NPCOL$ can be determined by calling the subroutine $\text{blacs_gridinfo}$.

### Output Parameters

$a$

On exit, if $\text{uplo} = 'U'$, the diagonal and first superdiagonal of sub($A$) are overwritten by the corresponding elements of the tridiagonal matrix $T$, and the elements above the first superdiagonal, with the array $\tau$, represent the orthogonal matrix $Q$ as a product of elementary reflectors; if $\text{uplo} = 'L'$, the diagonal and first subdiagonal of sub($A$) are overwritten by the corresponding elements of the tridiagonal matrix $T$, and the elements below the first subdiagonal, with the array $\tau$, represent the orthogonal matrix $Q$ as a product of elementary reflectors. See Further Details.

$d$ (local)

Array, size LOCc$(ja+n-1)$

The diagonal elements of the tridiagonal matrix $T$: $d(i) = A(i,i)$. $d$ is tied to the distributed matrix $A$.

$e$ (local)

Array, size LOCc$(ja+n-1)$ if $\text{uplo} = 'U'$, LOCc$(ja+n-2)$ otherwise.
The off-diagonal elements of the tridiagonal matrix $T$: $e(i) = A(i,i+1)$ if $uplo = 'U'$, $e(i) = A(i+1,i)$ if $uplo = 'L'$. $e$ is tied to the distributed matrix $A$.

$tau$

(local)

Array, size $\text{LOCc}(ja+n-1)$.

This array contains the scalar factors $tau$ of the elementary reflectors. $tau$ is tied to the distributed matrix $A$.

$work$

(local)

Array, size $(lwork)$

On exit, $work[0]$ returns the optimal $lwork$.

$info$

(global)

$= 0$: successful exit

$< 0$: If the $i$-th argument is an array and the $j$-th entry had an illegal value, then $info = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $uplo = 'U'$, the matrix $Q$ is represented as a product of elementary reflectors $Q = H(n-1) . . . H(2) H(1)$.

Each $H(i)$ has the form

$$H(i) = I - \tau \cdot v \cdot v'$$

where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored in $A(ia:ia+i-2, ja+i)$, and $\tau$ in $\tau(ja+i-1)$.

If $uplo = 'L'$, the matrix $Q$ is represented as a product of elementary reflectors $Q = H(1) H(2) . . . H(n-1)$.

Each $H(i)$ has the form

$$H(i) = I - \tau \cdot v \cdot v'$$

where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and $\tau$ in $\tau(ja+i-1)$.

The contents of $\text{sub}( A )$ on exit are illustrated by the following examples with $n = 5$:

if $uplo = 'U'$:

$$\begin{bmatrix}
  d & e & v_2 & v_3 & v_4 \\
  d & e & v_3 & v_4 \\
  d & e & v_3 \\
  d & e \\
  d
\end{bmatrix}$$

if $uplo = 'L'$:

$$\begin{bmatrix}
  d \\
  e & d \\
  v_1 & e & d \\
  v_1 & v_2 & e & d \\
  v_1 & v_2 & v_3 & e & d
\end{bmatrix}$$
where $d$ and $e$ denote diagonal and off-diagonal elements of $T$, and $v_l$ denotes an element of the vector defining $H(l)$.

**Alignment requirements**

The distributed submatrix sub$(A)$ must verify some alignment properties, namely the following expression should be true:

\[
( mb_a = nb_a \text{ and } IROFFA = ICOFFA \text{ and } IROFFA = 0 ) \text{ with } IROFFA = \text{mod}( ia-1, mb_a ), \text{ and } ICOFFA = \text{mod}( ja-1, nb_a ).
\]

$p?sytrd$

*Reduces a symmetric matrix to real symmetric tridiagonal form by an orthogonal similarity transformation.*

**Syntax**

```c
void pssytrd (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *d , float *e , float *tau , float *work , MKL_INT *lwork , MKL_INT *info );
void pdsytrd (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *d , double *e , double *tau , double *work , MKL_INT *lwork , MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**

The $p?sytrd$ function reduces a real symmetric matrix sub$(A)$ to symmetric tridiagonal form $T$ by an orthogonal similarity transformation:

\[ Q^* \text{sub}(A) Q = T, \]

where sub$(A) = A(ia:ia+n-1,ja:ja+n-1)$.

**Input Parameters**

- **uplo**
  
  (global)
  
  Specifies whether the upper or lower triangular part of the symmetric matrix sub$(A)$ is stored:
  
  If $uplo = 'U'$, upper triangular
  
  If $uplo = 'L'$, lower triangular

- **n**
  
  (global) The order of the distributed matrix sub$(A)$ ($n \geq 0$).

- **a**
  
  (local)
  
  Pointer into the local memory to an array of size $lld_a*\text{LOCc}(ja+n-1)$. On entry, this array contains the local pieces of the symmetric distributed matrix sub$(A)$.
  
  If $uplo = 'U'$, the leading $n$-by-$n$ upper triangular part of sub$(A)$ contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( sub(A) \) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. See Application Notes below.

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( desca \) (global and local) array of size \( dlen_\_ \). The array descriptor for the distributed matrix \( A \).

\( work \) (local) Workspace array of size \( lwork \).

\( lwork \) (local or global) size of \( work \), must be at least:

\[
lwork \geq \max(NB\cdot(np +1), 3\cdot NB),
\]

where \( NB = mb_\_a = nb_\_a \),

\[
np = \text{numroc}(n, NB, MYROW, iarow, NPROW),
\]

\[
 iarow = \text{indxg2p}(ia, NB, MYROW, rsrca_\_, NPROW).
\]

\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( MYROW, MYCOL, NPROW \) and \( NPCOL \) can be determined by calling the function \( \text{blacs_gridinfo} \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{p}x\text{erbla} \).

**Output Parameters**

\( a \)

On exit, if \( uplo = 'U' \), the diagonal and first superdiagonal of \( sub(A) \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements above the first superdiagonal, with the array \( tau \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors; if \( uplo = 'L' \), the diagonal and first subdiagonal of \( sub(A) \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements below the first subdiagonal, with the array \( tau \), represent the orthogonal matrix \( Q \) as a product of elementary reflectors. See Application Notes below.

\( d \) (local)

Arrays of size \( \text{LOCc}(ja+n-1) \). The diagonal elements of the tridiagonal matrix \( T \):

\[
d[i] = A(i+1,i+1), \ 0 \leq i < \text{LOCc}(ja+n-1).
\]

\( d \) is tied to the distributed matrix \( A \).

\( e \) (local)

Arrays of size \( \text{LOCc}(ja+n-1) \) if \( uplo = 'U' \), \( \text{LOCc}(ja+n-2) \) otherwise.

The off-diagonal elements of the tridiagonal matrix \( T \):

\[
e[i] = A(i+1,i+2), \ 0 \leq i < \text{LOCc}(ja+n-1) \text{ if } uplo = 'U',
\]

\[
e[i] = A(i+2,i+1) \text{ if } uplo = 'L'.
\]

\( e \) is tied to the distributed matrix \( A \).
tau

(local)

Arrays of size $\text{LOCc}(ja+n-1)$. This array contains the scalar factors of the
elementary reflectors. $\text{tau}$ is tied to the distributed matrix $A$.

work[0]

On exit $\text{work}[0]$ contains the minimum value of $lwork$ required for
optimum performance.

info

(global)

= 0: the execution is successful.

< 0: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had
an illegal value, then $\text{info} = -(i*100+j)$; if the $i$-th argument is a scalar
and had an illegal value, then $\text{info} = -i$.

Application Notes

If $\text{uplo} = 'U'$, the matrix $Q$ is represented as a product of elementary reflectors

$Q = H(n-1)... H(2) H(1)$.

Each $H(i)$ has the form

$H(i) = i - \text{tau} \times v \times v'$,

where $\text{tau}$ is a real scalar, and $v$ is a real vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and $\text{tau}$ in $\text{tau}[ja+i-2]$.

If $\text{uplo} = 'L'$, the matrix $Q$ is represented as a product of elementary reflectors

$Q = H(1) H(2)... H(n-1)$.

Each $H(i)$ has the form

$H(i) = i - \text{tau} \times v \times v'$,

where $\text{tau}$ is a real scalar, and $v$ is a real vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and $\text{tau}$ in $\text{tau}[ja+i-2]$.

The contents of sub($A$) on exit are illustrated by the following examples with $n = 5$:

If $\text{uplo} = 'U'$:

\[
\begin{bmatrix}
  d & e & v2 & v3 & v4 \\
  d & e & v3 & v4 \\
  d & e & v4 \\
  d & e \\
  d
\end{bmatrix}
\]

If $\text{uplo} = 'L'$:
where \( d \) and \( e \) denote diagonal and off-diagonal elements of \( T \), and \( v_i \) denotes an element of the vector defining \( H(i) \).

**See Also**

*Overview* for details of ScaLAPACK array descriptor structures and related notations.

**p?ormtr**

*Multiplying a general matrix by the orthogonal transformation matrix from a reduction to tridiagonal form determined by p?sytrd.*

**Syntax**

```c
void psormtr (char *side, char *uplo, char *trans, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);
void pdormtr (char *side, char *uplo, char *trans, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

This function overwrites the general real distributed \( m \)-by-\( n \) matrix \( sub(C) = C(ic:ic+m-1,jc:jc+n-1) \) with

- \( side = 'L' \) : \( Q^T \) or \( Q \) is applied from the left.
- \( side = 'R' \) : \( Q^T \) or \( Q \) is applied from the right.

where \( Q \) is a real orthogonal distributed matrix of order \( nq \), with \( nq = m \) if \( side = 'L' \) and \( nq = n \) if \( side = 'R' \).

\( Q \) is defined as the product of \( nq \) elementary reflectors, as returned by \( p?sytrd \).

If \( uplo = 'U' \), \( Q = H(nq-1)... H(2) \) \( H(1) \);
If \( uplo = 'L' \), \( Q = H(1) H(2)... H(nq-1) \).

**Input Parameters**

- \( side \) = \( 'L' \) : \( Q \) or \( Q^T \) is applied from the left.
- \( 'R' \) : \( Q \) or \( Q^T \) is applied from the right.
trans
(global)
= 'N', no transpose, Q is applied.
= 'T', transpose, QT is applied.

uplo
(global)
= 'U': Upper triangle of A(ia:*, ja:*) contains elementary reflectors
from p?sytrd;
= 'L': Lower triangle of A(ia:*, ja:*) contains elementary reflectors
from p?sytrd

m
(global) The number of rows in the distributed matrix sub(C) (m≥0).

n
(global) The number of columns in the distributed matrix sub(C) (n≥0).

a
(local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+m-1) if
side = 'L', and lld_a*LOCc(ja+n-1) if side = 'R'.
Contains the vectors that define the elementary reflectors, as returned by
p?sytrd.
If side='L', lld_a ≥ max(1,LOCr(ia+m-1));
If side = 'R', lld_a ≥ max(1, LOCr(ia+n-1)).

ia, ja
(global) The row and column indices in the global matrix A indicating the
first row and the first column of the submatrix A, respectively.
desca
(global and local) array of size dlen_. The array descriptor for the
distributed matrix A.
tau
(local)
Array of size of itau where
if side = 'L' and uplo = 'U', itau = LOCC(m_a),
if side = 'L' and uplo = 'L', itau = LOCC(ja+m-2),
if side = 'R' and uplo = 'U', itau = LOCC(n_a),
if side = 'R' and uplo = 'L', itau = LOCC(ja+n-2).

tau[i] must contain the scalar factor of the elementary reflector H(i+1), as
returned by p?sytrd (0 ≤ i < itau). tau is tied to the distributed matrix A.

c
(local)
Pointer into the local memory to an array of size lld_c*LOCc(jc+n-1).
Contains the local pieces of the distributed matrix sub (C).

ic, jc
(global) The row and column indices in the global matrix C indicating the
first row and the first column of the submatrix C, respectively.
desccc
(global and local) array of size dlen_. The array descriptor for the
distributed matrix C.
work
(local)
Workspace array of size lwork.
lwork

(local or global) size of work, must be at least:

if uplo = 'U',
    iaa = ia; jaa = ja+1, icc = ic; jcc = jc;
else uplo = 'L',
    iaa = ia+1, jaa = ja;
If side = 'L',
    icc = ic+1; jcc = jc;
else icc = ic; jcc = jc+1;
end if
end if
If side = 'L',
    mi= m-1; ni= n
    lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) + nb_a*nb_a
else
If side = 'R',
    mi= m; mi = n-1;
    lwork≥max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0+numroc(numroc(ni+icoffc, nb_a, 0, 0, NCOL), nb_a, 0, 0, lcmq), mpc0)))*nb_a) + nb_a*nb_a
end if

where lcmq = lcm/NPCOL with lcm = ilcm(NPROW, NCOL),
iroffa = mod(iaa-1, mb_a),
icoffa = mod(jaa-1, nb_a),
iarow = indxg2p(iaa, mb_a, MYROW, rsrc_a, NPROW),
npa0 = numroc(ni+iroffa, mb_a, MYROW, iarow, NPROW),
iroffc = mod(icc-1, mb_c),
icoffc = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrcc, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrcc, NCOL),
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NCOL),

NOTE

mod(x,y) is the integer remainder of x/y.

NOTE
mod(x,y) is the integer remainder of x/y.
ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo. If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

**Output Parameters**

- **c**: Overwritten by the product Q*sub(C), or Q'*sub(C), or sub(C)*Q', or sub(C)*Q.
- **work[0]**: On exit work[0] contains the minimum value of lwork required for optimum performance.
- **info** (global)
  - = 0: the execution is successful.
  - < 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?hengst**

Reduces a complex Hermitian-definite generalized eigenproblem to standard form.

**Syntax**

```c
void pchengst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_Complex8* b, const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, float* scale, MKL_Complex8* work, const MKL_INT* lwork, MKL_INT* info);
void pzhengst (const MKL_INT* ibtype, const char* uplo, const MKL_INT* n, MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_Complex16* b, const MKL_INT* ib, const MKL_INT* jb, const MKL_INT* descb, double* scale, MKL_Complex16* work, const MKL_INT* lwork, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

p?hengst reduces a complex Hermitian-definite generalized eigenproblem to standard form.

p?hengst performs the same function as p?hegst, but is based on rank 2K updates, which are faster and more scalable than triangular solves (the basis of p?hengst).


p?hengst also calls p?hegst when insufficient workspace is provided, hence p?hengst provides improved performance only when lwork is sufficient (as described in the parameter descriptions).
In the following sub( A ) denotes the submatrix A( ia:ia+n-1, ja:ja+n-1 ) and sub( B ) denotes the submatrix B( ib:ib+n-1, jb:jb+n-1 ).

If ibtype = 1, the problem is sub( A )*x = lambda*sub( B )*x, and sub( A ) is overwritten by inv(U)*sub( A )*inv(U) or inv(L)*sub( A )*inv(L).

If ibtype = 2 or 3, the problem is sub( A )*sub( B )*x = lambda*x or sub( B )*sub( A )*x = lambda*x, and sub( A ) is overwritten by U*sub( A )*U or L*sub( A )*L.

sub( B ) must have been previously factorized as U*U or L*L by p?potrf.

Input Parameters

ibtype (global)
1: compute inv(U)*sub( A )*inv(U) or inv(L)*sub( A )*inv(L);
2 or 3: compute U*sub( A )*U or L*sub( A )*L.

uplo (global)
'U': Upper triangle of sub( A ) is stored and sub( B ) is factored as U*U;
'L': Lower triangle of sub( A ) is stored and sub( B ) is factored as L*L.

n (global)
The order of the matrices sub( A ) and sub( B ). n >= 0.

a (local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1).

On entry, this array contains the local pieces of the n-by-n Hermitian distributed matrix sub( A ). If uplo = 'U', the leading n-by-n upper triangular part of sub( A ) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If uplo = 'L', the leading n-by-n lower triangular part of sub( A ) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global)
Global row index of matrix A, which points to the beginning of the submatrix on which to operate.

ja (global)
Global column index of matrix A, which points to the beginning of the submatrix on which to operate.

desca (global and local)
Array of size dlen_.
The array descriptor for the distributed matrix A.

b (local)
Pointer into the local memory to an array of size lld_b*LOCc(jb+n-1).

ib (global)
Global row index of matrix B, which points to the beginning of the submatrix on which to operate.
jb

(global)
Global column index of matrix B, which points to the beginning of the submatrix on which to operate.

descb

(global and local)
Array of size dlen_

The array descriptor for the distributed matrix B.

work

(local)
Array, size (lwork)

On exit, work(1) returns the minimal and optimal lwork.

lwork

(local)
The size of the array work.

lwork is local input and must be at least lwork >= MAX( NB * ( NP0 +1 ), 3 * NB ).

When ibtype = 1 and uplo = 'L', p?hentnd provides improved performance when lwork >= 2 * NP0 * NB + NQ0 * NB + NB * NB, where NB = mb_a = nb_a, NP0 = numroc( n, NB, 0, 0, NPROW ), NQ0 = numroc( n, NB, 0, 0, NPROW ), and numroc is a ScaLAPACK tool function.

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the subroutine blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the routine only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

a

On exit, if info = 0, the transformed matrix, stored in the same format as sub( A ).

scale

(global)
Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this routine.

scale is always returned as 1.0.

work

On exit, work[0] returns the minimal and optimal lwork.

info

(global)

= 0: successful exit
< 0: If the i-th argument is an array and the j-entry had an illegal value, then info = -(i*100+j), if the i-th argument is a scalar and had an illegal value, then info = -i.

p?hentrd

Reduces a complex Hermitian matrix to Hermitian tridiagonal form.
Syntax

void pchentrd (const char* uplo, const MKL_INT* n, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, float* d, float* e, MKL_Complex8* tau, MKL_Complex8* work, const MKL_INT* lwork, float* rwork, const MKL_INT* lrwork, MKL_INT* info);

void pzhentrd (const char* uplo, const MKL_INT* n, MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, double* d, double* e, MKL_Complex16* tau, MKL_Complex16* work, const MKL_INT* lwork, double* rwork, const MKL_INT* lrwork, MKL_INT* info);

Include Files

• mkl_scalapack.h

Description

p?hentrd is a prototype version of p?hetrd which uses tailored codes (either the serial, ?hetrd, or the parallel code, p?hettrd) when adequate workspace is provided.

p?hentrd reduces a complex Hermitian matrix sub( A ) to Hermitian tridiagonal form T by an unitary similarity transformation:

Q^* * sub( A ) * Q = T, where sub( A ) = A(ia:ia+n-1,ja:ja+n-1).

p?hentrd is faster than p?hetrd on almost all matrices, particularly small ones (i.e. n < 500 * sqrt(P) ), provided that enough workspace is available to use the tailored codes.

The tailored codes provide performance that is essentially independent of the input data layout.

The tailored codes place no restrictions on ia, ja, MB or NB. At present, ia, ja, MB and NB are restricted to those values allowed by p?hetrd to keep the interface simple (see the Application Notes section for more information about the restrictions).

Input Parameters

uplo (global)

Specifies whether the upper or lower triangular part of the Hermitian matrix sub( A ) is stored:

= 'U': Upper triangular

= 'L': Lower triangular

n (global)

The number of rows and columns to be operated on, i.e. the order of the distributed submatrix sub( A ). n >= 0.

a (local)

Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1).

On entry, this array contains the local pieces of the Hermitian distributed matrix sub( A ). If uplo = 'U', the leading n-by-n upper triangular part of sub( A ) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If uplo = 'L', the leading n-by-n lower triangular part of sub( A ) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

ia (global)
The row index in the global array $a$ indicating the first row of $\text{sub}(A)$.

$ja$ (global)

The column index in the global array $a$ indicating the first column of $\text{sub}(A)$.

$desca$ (global and local)

Array of size $\text{dlen}_\text{a}$.

The array descriptor for the distributed matrix $A$.

$work$ (local)

Array, size (lwork)

$lwork$ (local or global)

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq \text{MAX}( \text{NB} \times (\text{NP} + 1), 3 \times \text{NB} )$.

For optimal performance, greater workspace is needed:

$lwork \geq 2 \times (\text{ANB}+1) \times (4 \times \text{NPS}+2) + (\text{NPS}+4) \times \text{NPS}$

$\text{ANB} = \text{pjlaenv}( \text{ICTXT}, 3, 'p?hettrd', 'L', 0, 0, 0, 0 )$

$\text{ICTXT} = \text{desca}( \text{ctxt}_\text{a} )$

$\text{SQNPC} = \text{INT}( \sqrt{\text{REAL}(\text{NPROW} \times \text{NPCOL})}) )$

$\text{NPS} = \text{MAX}( \text{numroc}( n, 1, 0, 0, \text{SQNPC}), 2 \times \text{ANB} )$

numroc is a ScaLAPACK tool function.

pjlaenv is a ScaLAPACK environmental inquiry function.

$\text{NPROW}$ and $\text{NPCOL}$ can be determined by calling the subroutine blacs_gridinfo.

$rwork$ (local)

Array, size (lrwork)

$lrwork$ (local or global)

The size of the array $rwork$.

$lrwork$ is local input and must be at least $lrwork \geq 1$.

For optimal performance, greater workspace is needed, i.e. $lrwork \geq \text{MAX}(2 \times n)$

Output Parameters

$a$

Output on exit, if $\text{uplo} = 'U'$, the diagonal and first superdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal matrix $T$, and the elements above the first superdiagonal, with the array $\text{tau}$, represent the unitary matrix $Q$ as a product of elementary reflectors; if $\text{uplo} = 'L'$, the diagonal and first subdiagonal of $\text{sub}(A)$ are overwritten by the corresponding elements of the tridiagonal...
matrix $T$, and the elements below the first subdiagonal, with the array $	au$, represent the unitary matrix $Q$ as a product of elementary reflectors. See Application Notes.

d
(local)
Array, size $\text{LOCc}(ja+n-1)$

The diagonal elements of the tridiagonal matrix $T$: $d[i-1] = A(i,i)$. $d$ is tied to the distributed matrix $A$.

e
(local)
Array, size $\text{LOCc}(ja+n-1)$ if $\text{uplo} = 'U'$, $\text{LOCc}(ja+n-2)$ otherwise.

The off-diagonal elements of the tridiagonal matrix $T$: $e[i-1] = A(i,i+1)$ if $\text{uplo} = 'U'$, $e[i-1] = A(i+1,i)$ if $\text{uplo} = 'L'$. $e$ is tied to the distributed matrix $A$.

$\tau$
(local)
Array, size $\text{LOCc}(ja+n-1)$.

This array contains the scalar factors $\tau$ of the elementary reflectors. $\tau$ is tied to the distributed matrix $A$.

work
On exit, $\text{work}[0]$ returns the optimal $lwork$.

rwork
On exit, $\text{rwork}[0]$ returns the optimal $lrwork$.

info
(global)

$= 0$: successful exit

$< 0$: If the $i$-th argument is an array and the $j$-th entry had an illegal value, then $\text{info} = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

If $\text{uplo} = 'U'$, the matrix $Q$ is represented as a product of elementary reflectors

$$ Q = H(n-1) \ldots H(2) H(1). $$

Each $H(i)$ has the form

$$ H(i) = I - \tau \cdot v \cdot v', $$

where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2,ja+i)$, and $\tau$ in $\tau(ja+i-1)$.

If $\text{uplo} = 'L'$, the matrix $Q$ is represented as a product of elementary reflectors

$$ Q = H(1) H(2) \ldots H(n-1). $$

Each $H(i)$ has the form

$$ H(i) = I - \tau \cdot v \cdot v', $$

where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1,ja+i-1)$, and $\tau$ in $\tau(ja+i-1)$.

The contents of sub($A$) on exit are illustrated by the following examples with $n = 5$:

if $\text{uplo} = 'U'$:
where \( d \) and \( e \) denote diagonal and off-diagonal elements of \( T \), and \( vi \) denotes an element of the vector defining \( H(i) \).

**Alignment requirements**

The distributed submatrix \( \text{sub}(A) \) must verify some alignment properties, namely the following expression should be true:

\[
(mb_{-a} = nb_{-a} \text{ and } IROFFA = ICOFFA \text{ and } IROFFA = 0) \text{ with } IROFFA = \text{mod}(ia-1, mb_{-a}), \text{ and ICOFFA = mod}(ja-1, nb_{-a}).
\]

**p?hetrd**

*Reduces a Hermitian matrix to Hermitian tridiagonal form by a unitary similarity transformation.*

**Syntax**

```c
void pchetrd (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *d , float *e , MKL_Complex8 *tau , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzhetrd (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *d , double *e , MKL_Complex16 *tau , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?hetrd function reduces a complex Hermitian matrix \( \text{sub}(A) \) to Hermitian tridiagonal form \( T \) by a unitary similarity transformation:

\[
Q^{*}\text{sub}(A)^{*}Q = T
\]

where \( \text{sub}(A) = A(ia:ia+n-1,ja:ja+n-1) \).

**Input Parameters**

- \( uplo \) (global)

  Specifies whether the upper or lower triangular part of the Hermitian matrix \( \text{sub}(A) \) is stored:

  If \( uplo = 'U' \), upper triangular
If uplo = 'L', lower triangular

\( n \)  
(global) The order of the distributed matrix sub(\( A \)) (\( n \geq 0 \)).

\( a \)  
(local)
Pointer into the local memory to an array of size \( lld_a \ast \text{LOCc}(ja+n-1) \). On entry, this array contains the local pieces of the Hermitian distributed matrix sub(\( A \)).

If uplo = 'U', the leading \( n \)-by-\( n \) upper triangular part of sub(\( A \)) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If uplo = 'L', the leading \( n \)-by-\( n \) lower triangular part of sub(\( A \)) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced. (see Application Notes below).

\( ia, ja \)  
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( desca \)  
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( work \)  
(local)
Workspace array of size \( lwork \).

\( lwork \)  
(local or global) size of \( work \), must be at least:
\[ lwork \geq \max(NB\ast(np +1), 3\ast NB) \]

where \( NB = mb_a = nb_a \),
\( np = \text{numroc}(n, NB, MYROW, iarow, NPROW) \),
\( iarow = \text{indxg2p}(ia, NB, MYROW, rsrca, NPROW) \).

\text{indxg2p} and \text{numroc} are \text{ScaLAPACK} tool functions; \text{MYROW}, MYCOL, NPROW and NPCOL can be determined by calling the function \text{blacs_gridinfo}.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

**Output Parameters**

\( a \)  
On exit,
If uplo = 'U', the diagonal and first superdiagonal of sub(\( A \)) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements above the first superdiagonal, with the array \( tau \), represent the unitary matrix \( Q \) as a product of elementary reflectors; if uplo = 'L', the diagonal and first subdiagonal of sub(\( A \)) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements below the first subdiagonal, with the array \( tau \), represent the unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\( d \)  
(local)
Arrays of size $\text{LOC}(ja+n-1)$. The diagonal elements of the tridiagonal matrix $T$:
\[ d[i] = A(i+1,i+1), \quad 0 \leq i < \text{LOC}(ja+n-1). \]
$d$ is tied to the distributed matrix $A$.

$e$

(local)

Arrays of size $\text{LOC}(ja+n-1)$ if $\text{uplo} = 'U'$; $\text{LOC}(ja+n-2)$ otherwise. The off-diagonal elements of the tridiagonal matrix $T$:
\[ e[i] = A(i+1,i+2), \quad 0 \leq i < \text{LOC}(ja+n-1) \]
for $\text{uplo} = 'U'$,
\[ e[i] = A(i+2,i+1) \]
otherwise. $e$ is tied to the distributed matrix $A$.

$\tau$

(local)

Array of size $\text{LOC}(ja+n-1)$. This array contains the scalar factors of the elementary reflectors. $\tau$ is tied to the distributed matrix $A$.

$\text{work}[0]$

On exit $\text{work}[0]$ contains the minimum value of $l\text{work}$ required for optimum performance.

$info$

(global)

$= 0$: the execution is successful.

$< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

If $\text{uplo} = 'U'$, the matrix $Q$ is represented as a product of elementary reflectors
\[ Q = H(n-1)*...*H(2)*H(1). \]
Each $H(i)$ has the form
\[ H(i) = i - \tau*v*v', \]
where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(i+1:n) = 0$ and $v(i) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia+i-2, ja+i)$, and $\tau$ in $\tau[ja+i-2]$.

If $\text{uplo} = 'L'$, the matrix $Q$ is represented as a product of elementary reflectors
\[ Q = H(1)*H(2)*...*H(n-1). \]
Each $H(i)$ has the form
\[ H(i) = i - \tau*v*v', \]
where $\tau$ is a complex scalar, and $v$ is a complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2:n)$ is stored on exit in $A(ia+i+1:ia+n-1, ja+i-1)$, and $\tau$ in $\tau[ja+i-2]$.

The contents of sub($A$) on exit are illustrated by the following examples with $n = 5$:

If $\text{uplo} = 'U'$:
If \texttt{uplo} = 'L':

\[
\begin{bmatrix}
  d & e & v_2 & v_3 & v_4 \\
  & d & e & v_3 & v_4 \\
  & & d & e & v_4 \\
  & & & d & e \\
\end{bmatrix}
\]

where \(d\) and \(e\) denote diagonal and off-diagonal elements of \(T\), and \(v_i\) denotes an element of the vector defining \(H(i)\).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?unmtr**

*Multiplies a general matrix by the unitary transformation matrix from a reduction to tridiagonal form determined by p?hetrd.*

**Syntax**

```c
void pcunmtr (char *side, char *uplo, char *trans, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzunmtr (char *side, char *uplo, char *trans, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

This function overwrites the general complex distributed \(m\)-by-\(n\) matrix \(\text{sub}(C) = C(ic:ic+m-1,jc:jc+n-1)\) with

- \(side = 'L'\): \(Q\text{sub}(C)\)
- \(side = 'R'\): \(\text{sub}(C)Q\)
trans = 'C': Q^H*sub(C) sub(C)*Q^H

where Q is a complex unitary distributed matrix of order nq, with nq = m if side = 'L' and nq = n if side = 'R'.

Q is defined as the product of nq-1 elementary reflectors, as returned by p?hetrd.

If uplo = 'U', Q = H(nq-1)...H(2) H(1);
If uplo = 'L', Q = H(1) H(2)...H(nq-1).

Input Parameters

side (global)
= 'L': Q or Q^H is applied from the left.
= 'R': Q or Q^H is applied from the right.

trans (global)
= 'N', no transpose, Q is applied.
= 'C', conjugate transpose, Q^H is applied.

uplo (global)
= 'U': Upper triangle of A(ia:*, ja:*) contains elementary reflectors from p?hetrd;
= 'L': Lower triangle of A(ia:*, ja:) contains elementary reflectors from p?hetrd

m (global) The number of rows in the distributed matrix sub(C) (m≥0).

n (global) The number of columns in the distributed matrix sub(C) (n≥0).

a (local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+m-1) if side = 'L', and lld_a*LOCc(ja+m-1) if side = 'R'.

Contains the vectors which define the elementary reflectors, as returned by p?hetrd.

If side='L', lld_a ≥ max(1,LOCr(ia+m-1));
If side = 'R', lld_a ≥ max(1,LOCr(ia+n-1)).

ia, ja (global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

tau (local)
Array of size of ltau where
If side = 'L' and uplo = 'U', ltau = LOCc(m_a),
if side = 'L' and uplo = 'L', ltau = LOCc(ja+m-2),
if side = 'R' and uplo = 'U', ltau = LOCc(n_a),
if side = 'R' and uplo = 'L', ltau = LOCc(ja+n-2).
$\tau[i]$ must contain the scalar factor of the elementary reflector $H(i+1)$, as returned by $p\text{hetrd}$ ($0 \leq i < \ell\tau$). $\tau$ is tied to the distributed matrix $A$.

declaration(c)
Pointer into the local memory to an array of size $lld_c*\text{LOCc}(jc+n-1)$.
Contains the local pieces of the distributed matrix sub $(C)$.

declaration(ic, jc)
(global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix $C$, respectively.

declaration(desc)
(global and local) array of size $dlen_\text{.}$ The array descriptor for the distributed matrix $C$.

declaration(work)
(local)
Workspace array of size $lwork$.

declaration(lwork)
(local or global) size of $work$, must be at least:
If $\text{uplo} = 'U'$,
\begin{align*}
    iaa &= ia; jaa = ja+1, icc = ic; jcc = jc;
    \text{else } \text{uplo} &= 'L',
    iaa &= ia+1, jaa = ja;
    \text{If } \text{side} &= 'L',
    icc &= ic+1; jcc = jc;
    \text{else } icc &= ic; jcc = jc+1;
\end{align*}
end if
end if
If $\text{side} = 'L'$,
\begin{align*}
    mi &= m-1; ni = n;
    lwork &\geq \max((nb_a*(nb_a-1))/2, (nqc0 + mpc0)*nb_a) + nb_a*nb_a
    \text{else}
    lwork &\geq \max((nb_a*(nb_a-1))/2, (nqc0 + 
        \max(npa0+\text{numroc}(\text{numroc}(ni+\text{icoffc}, nb_a, 0, 0, \text{NPCOL}), nb_a, 0, 0, 1\text{lcmq}), mpc0))*nb_a) + nb_a*nb_a
    \end{align*}
end if
If $\text{side} = 'R'$,
\begin{align*}
    mi &= m; mi = n-1;
    lwork &\geq \max((nb_a*(nb_a-1))/2, (nqc0 + 
        \max(npa0+\text{numroc}(\text{numroc}(ni+\text{icoffc}, nb_a, 0, 0, \text{NPCOL}), nb_a, 0, 0, 1\text{lcmq}), mpc0))*nb_a) + nb_a*nb_a
    \end{align*}
end if
where $1\text{cmq} = \text{lcm}/\text{NPCOL}$ with $1\text{cm} = \text{ilcm}(\text{NPROW}, \text{NPCOL})$
\begin{align*}
    \text{irow} &= \text{indxg2p}(iaa, mb_a, \text{MYROW}, rsr\text{c}_a, \text{NPROW}),
    npa0 &= \text{numroc}(ni+\text{icoffa}, mb_a, \text{MYROW}, \text{irow}, \text{NPROW}),
    \text{iroffc} &= \text{mod}(\text{iccc}-1, \text{mb}_c),
\end{align*}
\( \text{icoffe} = \text{mod}(jcc - 1, \ nb_c), \)
\( \text{icrow} = \text{indxg2p}(icc, \ mb_c, \ MYROW, \ rsr_c_c, \ NPROW), \)
\( \text{iccol} = \text{indxg2p}(jcc, \ nb_c, \ MYCOL, \ csr_c_c, \ NPCOL), \)
\( \text{mpc0} = \text{numroc}(mi + iroffc, \ mb_c, \ MYROW, \ icrow, \ NPROW), \)
\( \text{nqc0} = \text{numroc}(ni + icoffc, \ nb_c, \ MYCOL, \ iccol, \ NPCOL), \)

**NOTE**

\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

\( \text{ilcm}, \ \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( \text{MYROW}, \ \text{MYCOL}, \ \text{NPROW} \) and \( \text{NPCOL} \) can be determined by calling the function \( \text{blacs_gridinfo} \). If \( \text{lwork} = -1 \), then \( \text{lwork} \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \).

**Output Parameters**

- \( c \)
  - Overwritten by the product \( Q \times \text{sub}(C), \) or \( Q' \times \text{sub}(C), \) or \( \text{sub}(C) \times Q', \) or \( \text{sub}(C) \times Q. \)
  - \( \text{work}[0] \)
    - On exit \( \text{work}[0] \) contains the minimum value of \( \text{lwork} \) required for optimum performance.
  - \( \text{info} \)
    - (global)
      - \( = 0: \) the execution is successful.
      - \( < 0: \) if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j - 1, \) had an illegal value, then \( \text{info} = -(i \times 100 + j); \) if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i. \)

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?stebz**

*Computes the eigenvalues of a symmetric tridiagonal matrix by bisection.*

**Syntax**

```c
void psstebz (MKL_INT *ictxt, char *range, char *order, MKL_INT *n, float *vl, float *vu, MKL_INT *il, MKL_INT *iu, float *abstol, float *d, float *e, MKL_INT *m, MKL_INT *nsplit, float *w, MKL_INT *iblock, MKL_INT *isplit, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *info);
void pdstebz (MKL_INT *ictxt, char *range, char *order, MKL_INT *n, double *vl, double *vu, MKL_INT *il, MKL_INT *iu, double *abstol, double *d, double *e, MKL_INT *m, MKL_INT *nsplit, double *w, MKL_INT *iblock, MKL_INT *isplit, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *info);
```
Include Files

- mkl_scalapack.h

Description

The \texttt{p?stebz} function computes the eigenvalues of a symmetric tridiagonal matrix in parallel. These may be all eigenvalues, all eigenvalues in the interval \([vl, vu]\), or the eigenvalues \(il\) through \(iu\). A static partitioning of work is done at the beginning of \texttt{p?stebz} which results in all processes finding an (almost) equal number of eigenvalues.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\begin{itemize}
  \item \texttt{ictxt} (global) The BLACS context handle.
  \item \texttt{range} (global) Must be \texttt{'A'} or \texttt{'V'} or \texttt{'I'}.
    \begin{itemize}
      \item If \texttt{range = 'A'}, the function computes all eigenvalues.
      \item If \texttt{range = 'V'}, the function computes eigenvalues in the interval \([vl, vu]\).
      \item If \texttt{range = 'I'}, the function computes eigenvalues \(il\) through \(iu\).
    \end{itemize}
  \item \texttt{order} (global) Must be \texttt{'B'} or \texttt{'E'}.
    \begin{itemize}
      \item If \texttt{order = 'B'}, the eigenvalues are to be ordered from smallest to largest within each split-off block.
      \item If \texttt{order = 'E'}, the eigenvalues for the entire matrix are to be ordered from smallest to largest.
    \end{itemize}
  \item \texttt{n} (global) The order of the tridiagonal matrix \(T(n \geq 0)\).
  \item \texttt{vl, vu} (global)
    \begin{itemize}
      \item If \texttt{range = 'V'}, the function computes the lower and the upper bounds for the eigenvalues on the interval \([l, u]\).
      \item If \texttt{range = 'A' or 'I'}, \(vl\) and \(vu\) are not referenced.
    \end{itemize}
  \item \texttt{il, iu} (global)
    \begin{itemize}
      \item Constraint: \(1 \leq il \leq iu \leq n\).
      \item If \texttt{range = 'I'}, the index of the smallest eigenvalue is returned for \(il\) and of the largest eigenvalue for \(iu\) (assuming that the eigenvalues are in ascending order) must be returned.
    \end{itemize}
\end{itemize}
If range = 'A' or 'V', il and iu are not referenced.

abstol

The absolute tolerance to which each eigenvalue is required. An eigenvalue (or cluster) is considered to have converged if it lies in an interval of width abstol. If abstol≤0, then the tolerance is taken as ulp||T||, where ulp is the machine precision, and ||T|| means the 1-norm of T.

Eigenvalues will be computed most accurately when abstol is set to the underflow threshold slamch('U'), not 0. Note that if eigenvectors are desired later by inverse iteration (p?stein), abstol should be set to 2*p?lamch('S').

d

Array of size n.

Contains n diagonal elements of the tridiagonal matrix T. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than overflow^{1/2} * underflow^{1/4} in absolute value, and for greatest accuracy, it should not be much smaller than that.

e

Array of size n - 1.

Contains (n-1) off-diagonal elements of the tridiagonal matrix T. To avoid overflow, the matrix must be scaled so that its largest entry is no greater than overflow^{1/2} * underflow^{1/4} in absolute value, and for greatest accuracy, it should not be much smaller than that.

work

Array of size max(5n, 7). This is a workspace array.

lwork

(local) The size of the work array must be ≥ max(5n, 7).

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

iwork

(local) Array of size max(4n, 14). This is a workspace array.

liwork

(local) The size of the iwork array must ≥max(4n, 14, NPROCS).

If liwork = -1, then liwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

m

(global) The actual number of eigenvalues found. 0≤m≤n

nsplit

(global) The number of diagonal blocks detected in T. 1≤nsplit≤n

w

(global)
Array of size \( n \). On exit, the first \( m \) elements of \( w \) contain the eigenvalues on all processes.

\( iblock \) (global)

Array of size \( n \). At each row/column \( j \) where \( e[j-1] \) is zero or small, the matrix \( T \) is considered to split into a block diagonal matrix. On exit \( iblock[i] \) specifies which block (from 1 to the number of blocks) the eigenvalue \( w[i] \) belongs to.

**NOTE**
In the (theoretically impossible) event that bisection does not converge for some or all eigenvalues, \( info \) is set to 1 and the ones for which it did not are identified by a negative block number.

\( isplit \) (global)

Array of size \( n \).

Contains the splitting points, at which \( T \) breaks up into submatrices. The first submatrix consists of rows/columns 1 to \( isplit[0] \), the second of rows/columns \( isplit[0]+1 \) through \( isplit[1] \), and so on, and the \( nsplit \)-th submatrix consists of rows/columns \( isplit[nsplit-2]+1 \) through \( isplit[nsplit-1]=n \). (Only the first \( nsplit \) elements are used, but since the \( nsplit \) values are not known, \( n \) words must be reserved for \( isplit \).)

\( info \) (global)

If \( info = 0 \), the execution is successful.

If \( info < 0 \), if \( info = -i \), the \( i \)-th argument has an illegal value.

If \( info > 0 \), some or all of the eigenvalues fail to converge or are not computed.

If \( info = 1 \), bisection fails to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.

If \( info = 2 \), mismatch between the number of eigenvalues output and the number desired.

If \( info = 3 \): range='I', and the Gershgorin interval initially used is incorrect. No eigenvalues are computed. Probable cause: the machine has a sloppy floating-point arithmetic. Increase the \( fudge \) parameter, recompile, and try again.

**See Also**
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?stedc
Computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel.
Syntax

void psstedc (const char* compz, const MKL_INT n, float* d, float* e, float* q, const MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, float* work, MKL_INT* lwork, MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);

void pdstedc (const char* compz, const MKL_INT n, double* d, double* e, double* q, const MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, double* work, MKL_INT* lwork, MKL_INT* iwork, const MKL_INT* liwork, MKL_INT* info);

Include Files

• mkl_scalapack.h

Description

p?stedc computes all eigenvalues and eigenvectors of a symmetric tridiagonal matrix in parallel, using the divide and conquer algorithm.

Input Parameters

compz

= 'N': Compute eigenvalues only. (NOT IMPLEMENTED YET)
= 'I': Compute eigenvectors of tridiagonal matrix also.
= 'V': Compute eigenvectors of original dense symmetric matrix also. On entry, Z contains the orthogonal matrix used to reduce the original matrix to tridiagonal form. (NOT IMPLEMENTED YET)

n

(global)
The order of the tridiagonal matrix \( T \). \( n \geq 0 \).

d

(global)
Array, size \((n)\)
On entry, the diagonal elements of the tridiagonal matrix.

e

(global)
Array, size \((n-1)\).
On entry, the subdiagonal elements of the tridiagonal matrix.

iq

(global)
Q's global row index, which points to the beginning of the submatrix which is to be operated on.

jq

(global)
Q's global column index, which points to the beginning of the submatrix which is to be operated on.

descq

(global and local)
Array of size \( dlen_\).
The array descriptor for the distributed matrix \( Q \).

work

(local)
Array, size \((lwork)\)
**lwork**

(llocal)

The size of the array \( \text{work} \).

\[ lwork = 6*n + 2*NP*NQ \]

\[ NP = \text{numroc}( n, NB, MYROW, \text{DESCQ}( \text{rsrc}_\text{r}), \text{NPROW} ) \]

\[ NQ = \text{numroc}( n, NB, MYCOL, \text{DESCQ}( \text{csrc}_\text{c}), \text{NPCOL} ) \]

\text{numroc} \text{ is a ScaLAPACK tool function.}

If \( lwork = -1 \), the \( lwork \) is global input and a workspace query is assumed; the routine only calculates the minimum size for the \( \text{work} \) array. The required workspace is returned as the first element of \( \text{work} \) and no error message is issued by \text{p}\text{xerbla}.

**iwork**

(llocal)

Array, size (\( liwork \))

**liwork**

The size of the array \( \text{iwork} \).

\[ liwork = 2 + 7*n + 8*\text{NPCOL} \]

**Output Parameters**

\( d \)  

On exit, if \( info = 0 \), the eigenvalues in descending order.

\( q \)  

(llocal)

Array, local size (\( lld_q \), \text{LOCc}(jq+n-1))

\( q \) contains the orthonormal eigenvectors of the symmetric tridiagonal matrix.

On output, \( q \) is distributed across the P processes in block cyclic format.

\( \text{work} \)  

On output, \( \text{work}[0] \) returns the workspace needed.

\( iwork \)  

On exit, if \( liwork > 0 \), \( iwork[0] \) returns the optimal \( liwork \).

\( info \)  

(global)

\( = 0 \): successful exit.

\( < 0 \): If the \( i \)-th argument is an array and the \( j \)-th entry had an illegal value, then \( info = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( > 0 \): The algorithm failed to compute the \( info/(n+1) \)-th eigenvalue while working on the submatrix lying in global rows and columns \( \text{mod}(info,n+1) \).

**p?stein**

Computes the eigenvectors of a tridiagonal matrix using inverse iteration.
Syntax

```c
void psstein (MKL_INT *n, float *d, float *e, MKL_INT *m, float *w, MKL_INT *iblock,
              MKL_INT *isplit, float *orfac, float *z, MKL_INT *iz, MKL_INT *jz,
              MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *iwork,
              MKL_INT *ifail, MKL_INT *iclustr, float *gap, MKL_INT *info);
void pdstein (MKL_INT *n, double *d, double *e, MKL_INT *m, double *w, MKL_INT *iblock,
              MKL_INT *isplit, double *orfac, double *z, MKL_INT *iz, MKL_INT *jz,
              MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *iwork,
              MKL_INT *ifail, MKL_INT *iclustr, double *gap, MKL_INT *info);
void pcstein (MKL_INT *n, float *d, float *e, MKL_INT *m, float *w, MKL_INT *iblock,
              MKL_INT *isplit, float *orfac, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz,
              MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *iwork,
              MKL_INT *ifail, MKL_INT *iclustr, float *gap, MKL_INT *info);
void pzstein (MKL_INT *n, double *d, double *e, MKL_INT *m, double *w, MKL_INT *iblock,
              MKL_INT *isplit, double *orfac, MKL_Complex16 *z, MKL_INT *iz, MKL_INT *jz,
              MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *iwork,
              MKL_INT *ifail, MKL_INT *iclustr, double *gap, MKL_INT *info);
```

Include Files

- mkl_scalapack.h

Description

The `p?stein` function computes the eigenvectors of a symmetric tridiagonal matrix \( T \) corresponding to specified eigenvalues, by inverse iteration. `p?stein` does not orthogonalize vectors that are on different processes. The extent of orthogonalization is controlled by the input parameter `lwork`. Eigenvectors that are to be orthogonalized are computed by the same process. `p?stein` decides on the allocation of work among the processes and then calls `?stein2` (modified LAPACK function) on each individual process. If insufficient workspace is allocated, the expected orthogonalization may not be done.

**NOTE**

If the eigenvectors obtained are not orthogonal, increase `lwork` and run the code again.

\( p = \text{NPROW} \times \text{NPCOL} \) is the total number of processes.

**Input Parameters**

- \( n \) (global) The order of the matrix \( T(n \geq 0) \).
- \( m \) (global) The number of eigenvectors to be returned.
- \( d, e, w \) (global) Arrays:
  - \( d \) of size \( n \) contains the diagonal elements of \( T \).
  - \( e \) of size \( n-1 \) contains the off-diagonal elements of \( T \).
  - \( w \) of size \( m \) contains all the eigenvalues grouped by split-off block. The eigenvalues are supplied from smallest to largest within the block. (Here the output array \( w \) from `p?stebz` with order = 'B' is expected. The array should be replicated in all processes.)
iblock

(global)
Array of size n. The submatrix indices associated with the corresponding
eigenvalues in w: 1 for eigenvalues belonging to the first submatrix from
the top, 2 for those belonging to the second submatrix, etc. (The output
array iblock from p?stebz is expected here).

isplit

(global)
Array of size n. The splitting points at which T breaks up into submatrices.
The first submatrix consists of rows/columns 1 to isplit[0], the second of
rows/columns isplit[0]+1 through isplit[1], and so on, and the
nsplit-th submatrix consists of rows/columns isplit[nsplit-2]+1
through isplit[nsplit-1]=n. (The output array isplit from p?stebz is
expected here.)

orfac

(global)
orfac specifies which eigenvectors should be orthogonalized. Eigenvectors
that correspond to eigenvalues within orfac*||T|| of each other are to be
orthogonalized. However, if the workspace is insufficient (see lwork), this
tolerance may be decreased until all eigenvectors can be stored in one
process. No orthogonalization is done if orfac is equal to zero. A default
value of 1000 is used if orfac is negative. orfac should be identical on all
processes

iz, jz

(global) The row and column indices in the global matrix Z indicating the
first row and the first column of the submatrix Z, respectively.

descz

(global and local) array of size dlen_. The array descriptor for the
distributed matrix Z.

work

(local).
Workspace array of size lwork.

lwork

(local)
lwork controls the extent of orthogonalization which can be done. The
number of eigenvectors for which storage is allocated on each process is
nvec = floor((lwork-max(5*n,np00*mq00))/n). Eigenvectors
corresponding to eigenvalue clusters of size (nvec - ceil(m/p) + 1) are
guaranteed to be orthogonal (the orthogonality is similar to that obtained
from ?stein2).

NOTE

lwork must be no smaller than max(5*n,np00*mq00) +
ceil(m/p)*n and should have the same input value on all
processes.

It is the minimum value of lwork input on different processes that is
significant.
If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

**iwork**

(local)
Workspace array of size $3n+p+1$.

**liwork**

(local) The size of the array `iwork`. It must be greater than $3n+p+1$.
If $liwork = -1$, then `liwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

**Output Parameters**

**z**

(local)
Array of size `descz[dlen_-1], n/NPCOL + NB`). `z` contains the computed eigenvectors associated with the specified eigenvalues. Any vector which fails to converge is set to its current iterate after MAXIT iterations (See `?stein2`). On output, `z` is distributed across the $p$ processes in block cyclic format.

**work**

On exit, `work[0]` gives a lower bound on the workspace (`lwork`) that guarantees the user desired orthogonalization (see `orfac`). Note that this may overestimate the minimum workspace needed.

**iwork**

On exit, `iwork[0]` contains the amount of integer workspace required.

On exit, the `iwork[1]` through `iwork[p+1]` indicate the eigenvectors computed by each process. Process $i$ computes eigenvectors indexed `iwork[i+1]+1` through `iwork[i+2]`.

**ifail**

(global) Array of size $m$. On normal exit, all elements of `ifail` are zero. If one or more eigenvectors fail to converge after MAXIT iterations (as in `?stein`), then `info > 0` is returned. If $\mod(info, m+1)>0$, then for $i=1$ to $\mod(info, m+1)$, the eigenvector corresponding to the eigenvalue $w[ifail[i-1]-1]$ failed to converge ($w$ refers to the array of eigenvalues on output).

**NOTE**

\( \mod(x, y) \) is the integer remainder of \(x/y\).

**iclustr**

(global) Array of size $2*p$.

This output array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be orthogonalized due to insufficient workspace (see `lwork`, `orfac` and `info`). Eigenvectors corresponding to clusters of eigenvalues indexed `iclustr(2*I-1)` to `iclustr(2*I)`, $i = 1$ to $\mod(info, m+1)$, could not be orthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. `iclustr` is a zero terminated array; `iclustr[2*k-1] ≠ 0` and `iclustr[2*k] = 0` if and only if $k$ is the number of clusters.
This output array contains the gap between eigenvalues whose eigenvectors could not be orthogonalized. The $\text{info}/m$ output values in this array correspond to the $\text{info}/(m+1)$ clusters indicated by the array $\text{iclustr}$. As a result, the dot product between eigenvectors corresponding to the $i$-th cluster may be as high as $(O(n)*\text{macheps})/\text{gap}[i-1]$.

If $\text{info} = 0$, the execution is successful.

If $\text{info} < 0$: If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $\text{info} = -(i*100+j)$.

If the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.

If $\text{info} < 0$: if $\text{info} = -i$, the $i$-th argument had an illegal value.

If $\text{info} > 0$: if $\text{mod}(\text{info}, m+1) = i$, then $i$ eigenvectors failed to converge in MAXIT iterations. Their indices are stored in the array $\text{ifail}$. If $\text{info}/(m+1) = i$, then eigenvectors corresponding to $i$ clusters of eigenvalues could not be orthogonalized due to insufficient workspace. The indices of the clusters are stored in the array $\text{iclustr}$.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

Nonsymmetric Eigenvalue Problems: ScaLAPACK Computational Routines
This section describes ScaLAPACK routines for solving nonsymmetric eigenvalue problems, computing the Schur factorization of general matrices, as well as performing a number of related computational tasks.

To solve a nonsymmetric eigenvalue problem with ScaLAPACK, you usually need to reduce the matrix to the upper Hessenberg form and then solve the eigenvalue problem with the Hessenberg matrix obtained.

Table "Computational Routines for Solving Nonsymmetric Eigenproblems" lists ScaLAPACK routines for reducing the matrix to the upper Hessenberg form by an orthogonal (or unitary) similarity transformation $A=QHQ^H$, as well as routines for solving eigenproblems with Hessenberg matrices, and multiplying the matrix after reduction.

Computational Routines for Solving Nonsymmetric Eigenproblems

<table>
<thead>
<tr>
<th>Operation performed</th>
<th>General matrix</th>
<th>Orthogonal/Unitary matrix</th>
<th>Hessenberg matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce to Hessenberg form $A=QHQ^H$</td>
<td>$\text{p?gehrd}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply the matrix after reduction</td>
<td>$\text{p?ormhr}$</td>
<td>$\text{p?unmhr}$</td>
<td>$\text{p?lahqr}$</td>
</tr>
<tr>
<td>Find eigenvalues and Schur factorization</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$p?gehrd$

Reduces a general matrix to upper Hessenberg form.

Syntax

```c
void psgehrd (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, float *a, MKL_INT *ia,
MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT
*info );
```
void pdgehrd (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);

void pcgehrd (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzgehrd (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?gehrd function reduces a real/complex general distributed matrix sub(A) to upper Hessenberg form $H$ by an orthogonal or unitary similarity transformation $Q^*\text{sub}(A)Q = H$, where sub(A) = $A(ia:ia+n-1, ja:ja+n-1)$.

Input Parameters

- **n** (global). The order of the distributed matrix sub(A) ($n \geq 0$).
- **ilo, ihi** (global). It is assumed that sub(A) is already upper triangular in rows $ia:ia+ilo-2$ and $ia+ihi:ia+n-1$ and columns $ja:ja+ilo-2$ and $ja+ihi:ja+n-1$. (See Application Notes below).
- **a** (local) Pointer into the local memory to an array of size $ldd_a*\text{LOCc}(ja+n-1)$. On entry, this array contains the local pieces of the $n$-by-$n$ general distributed matrix sub(A) to be reduced.
- **ia, ja** (global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix $A$.
- **desca** (global and local) array of size $dlen_$. The array descriptor for the distributed matrix $A$.
- **work** (local) Workspace array of size $lwork$.
- **lwork** (local or global) size of the array $\text{work}$. $lwork$ is local input and must be at least

\[ lwork \geq \text{NB} \times \text{NB} + \text{NB} \times \max(ihip+1, ihlp+inlq) \]

where $\text{NB} = mb_a = nb_a$, $iroff_a = \text{mod}(ia-1, \text{NB})$, $icoff_a = \text{mod}(ja-1, \text{NB})$,
ioff = mod(ia+ilo-2, NB), iarow = indxg2p(ia, NB, MYROW, rsra, NPRW), ihip = numroc(ihi+iroff, NB, MYROW, iarow, NPRW),
ilrow = indxg2p(ia+ilo-1, NB, MYROW, rsra, NPRW),
ihlp = numroc(ihi-ilo+ioff+1, NB, MYROW, ilrow, NPRW),
ilcol = indxg2p(ja+ilo-1, NB, MYCOL, csrca, NPCOL),
inlq = numroc(n-ilo+ioff+1, NB, MYCOL, ilcol, NPCOL),

**NOTE**
mod(x, y) is the integer remainder of x/y.

indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPRW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

## Output Parameters

**a**

On exit, the upper triangle and the first subdiagonal of sub(A) are overwritten with the upper Hessenberg matrix $H$, and the elements below the first subdiagonal, with the array $tau$, represent the orthogonal/unitary matrix $Q$ as a product of elementary reflectors (see Application Notes below).

**tau**

(local).

Array of size at least max(ja+n-2).

The scalar factors of the elementary reflectors (see Application Notes below). Elements ja:ja+ilo-2 and ja+ihi:ja+n-2 of the global vector $tau$ are set to zero. $tau$ is tied to the distributed matrix $A$.

**work[0]**

On exit work[0] contains the minimum value of lwork required for optimum performance.

**info**

(global)

= 0: the execution is successful.

< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

## Application Notes

The matrix $Q$ is represented as a product of (ihi-ilo) elementary reflectors

$Q = H(ilo)*H(ilo+1)*...*H(ihi-1)$.

Each $H(i)$ has the form

$H(i) = i - tau*v*v'$
where \( \tau \) is a real/complex scalar, and \( \nu \) is a real/complex vector with \( \nu(1:) = 0, \nu(i+1) = 1 \) and \( \nu(ihi +1:n) = 0 \); \( \nu(i+2:ihi) \) is stored on exit in \( A(ia+ilo+1:ia+ihi-1, ja+ilo+i-2) \), and \( \tau \) in \( \tau[ja+ilo+i-3] \). The contents of \( A(ia:ia+n-1, ja:ja+n-1) \) are illustrated by the following example, with \( n = 7, ilo = 2 \) and \( ihi = 6 \):

on entry

\[
\begin{bmatrix}
a & a & a & a & a & a & a \\
ad & a & a & a & a & a \\
ad & d & a & a & a & a \\
d & d & d & a & a & a \\
a & a & a & a & a & a \\
a & d & a & a & a & a \\
a & d & d & d & d & a \\
\end{bmatrix}
\]

on exit

\[
\begin{bmatrix}
a & a & a & h & h & h & a \\
ad & h & h & h & h & h & a \\
h & h & h & h & h & h \\
v2 & h & h & h & h \\
v2 & v3 & h & h & h \\
v2 & v3 & v4 & h & h \\
\end{bmatrix}
\]

where \( a \) denotes an element of the original matrix \( \text{sub}(A) \), \( H \) denotes a modified element of the upper Hessenberg matrix \( H \), and \( vi \) denotes an element of the vector defining \( H(ja+ilo+i-2) \).

**See Also**

See Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ormhr**

Multiplies a general matrix by the orthogonal transformation matrix from a reduction to Hessenberg form determined by \( p?gehrd \).

**Syntax**

```c
void psormhr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info );
```
void pdormhr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The pormhr function overwrites the general real distributed m-by-n matrix sub(C) = C(ic:ic+m-1,ja:ja+n-1) with

\[
\begin{align*}
\text{side} = 'L' & \quad \text{side} = 'R' \\
\text{trans} = 'N' & \quad Q \cdot \text{sub}(C) & \quad \text{sub}(C) \cdot Q \\
\text{trans} = 'T' & \quad Q^T \cdot \text{sub}(C) & \quad \text{sub}(C) \cdot Q^T
\end{align*}
\]

where Q is a real orthogonal distributed matrix of order \(n_q\), with \(n_q = m\) if \(\text{side} = 'L'\) and \(n_q = n\) if \(\text{side} = 'R'\).

Q is defined as the product of \(ihi-ilo\) elementary reflectors, as returned by pgehrd.

\[Q = H(ilo) H(ilo+1) \ldots H(ihi-1)\]

Input Parameters

\textit{side} (global)
- '=L': Q or \(Q^T\) is applied from the left.
- '=R': Q or \(Q^T\) is applied from the right.

\textit{trans} (global)
- '=N', no transpose, Q is applied.
- '=T', transpose, \(Q^T\) is applied.

\textit{m} (global) The number of rows in the distributed matrix sub(C) (\(m \geq 0\)).

\textit{n} (global) The number of columns in the distributed matrix sub(C) (\(n \geq 0\)).

\textit{ilo, ihi} (global)
\- \textit{ilo} and \textit{ihi} must have the same values as in the previous call of pgehrd.
\- Q is equal to the unit matrix except for the distributed submatrix \(Q(ia+ilo:ihi-1,ja+ilo:ja+ihi-1)\).
\- If \textit{side} = 'L', \(1 \leq ilo \leq ihi \leq \max(1,m)\);
\- If \textit{side} = 'R', \(1 \leq ilo \leq ihi \leq \max(1,n)\);
\- \textit{ilo} and \textit{ihi} are relative indexes.

\textit{a} (local)
\- Pointer into the local memory to an array of size \(lld_a*\text{LOC}c(ja+m-1)\) if \textit{side} = 'L', and \(lld_a*\text{LOC}c(ja+n-1)\) if \textit{side} = 'R'.
\- Contains the vectors which define the elementary reflectors, as returned by pgehrd.
\(ia, ja\) (global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.

\(desca\) (global and local) array of size \(dlen_.\) The array descriptor for the distributed matrix \(A\).

\(tau\) (local)

Array of size \(\text{LOC}(ja+m-2)\) if \(side = 'L'\), and \(\text{LOC}(ja+n-2)\) if \(side = 'R'\).

\(\tau[j]\) contains the scalar factor of the elementary reflector \(H(j+1)\) as returned by \(p?gehrd\) \((0 \leq j < \text{size}(\tau))\). \(\tau\) is tied to the distributed matrix \(A\).

\(c\) (local)

Pointer into the local memory to an array of size \(\text{lld}_c^{*}\text{LOC}(jc+n-1)\).

Contains the local pieces of the distributed matrix \(\text{sub}(C)\).

\(ic, jc\) (global) The row and column indices in the global matrix \(C\) indicating the first row and the first column of the submatrix \(C\), respectively.

\(desccc\) (global and local) array of size \(dlen_.\) The array descriptor for the distributed matrix \(C\).

\(work\) (local)

Workspace array with size \(lwork\).

\(lwork\) (local or global)

The size of the array \(work\).

\(lwork\) must be at least \(iaa = ia + ilo; jaa = ja+ilo-1;\)

If \(side = 'L'\),

\(mi = ihi-ilo; ni = n; icc = ic + ilo; jcc = jc; lwork \geq \max((\text{nb}_a*(\text{nb}_a-1))/2, (\text{nqc}_0+\text{mpc}_0)*\text{nb}_a) + \text{nb}_a*\text{nb}_a\)

else if \(side = 'R'\),

\(mi = m; ni = ihi-ilo; icc = ic; jcc = jc + ilo; lwork \geq \max((\text{nb}_a*(\text{nb}_a-1))/2, (\text{nqc}_0+\max(\text{npa}_0+\text{numroc}(\text{numroc}(ni + \text{icoff}_c, \text{nb}_a, 0, 0, \text{NPCOL}), \text{nb}_a, 0, 0, \text{lcm}_q), \text{mpc}_0))*\text{nb}_a) + \text{nb}_a*\text{nb}_a\)

end if

where \(\text{lcm}_q = \text{lcm}/\text{NPCOL}\) with \(\text{lcm} = \text{ilcm}(\text{NPROW}, \text{NFCOL})\),

\(\text{irowf} = \text{mod}(iaa-1, \text{mb}_a),\)

\(\text{icoff}_a = \text{mod}(jaa-1, \text{nb}_a),\)

\(\text{iarow} = \text{indxg2p}(iaa, \text{mb}_a, \text{MYROW}, \text{rsr}_c, \text{NPROW}),\)

\(\text{npa}_0 = \text{numroc}(ni + \text{icoff}_a, \text{mb}_a, \text{MYROW}, \text{iarow}, \text{NPROW}),\)

\(\text{icoff}_c = \text{mod}(icc-1, \text{mb}_c), \text{icoff}_c = \text{mod}(jcc-1, \text{nb}_c),\)

\(\text{icrow} = \text{indxg2p}(icc, \text{mb}_c, \text{MYROW}, \text{rsr}_c, \text{NPROW}),\)

\(\text{icol} = \text{indxg2p}(jcc, \text{nb}_c, \text{MYCOL}, \text{csr}_c, \text{NPCOL}),\)
mpc0 = numroc(mi+iroffc, mb_c, MYROW, icrow, NPROW),

nqc0 = numroc(ni+icoffc, nb_c, MYCOL, iccol, NPCOL),

NOTE

mod(x,y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

Sub(C) is overwritten by Q*sub(C), or Q'*sub(C), or sub(C)*Q', or sub(C)*Q.

work[0] On exit work[0] contains the minimum value of lwork required for optimum performance.

info (global) = 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -(i*100+j); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?unmhr
Multiplies a general matrix by the unitary transformation matrix from a reduction to Hessenberg form determined by p?gehrd.

Syntax

void pcunmhr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzunmhr (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

• mkl_scalapack.h
Description

This function overwrites the general complex distributed $m$-by-$n$ matrix $\text{sub}(C) = C(i_c:i_c+m-1,j_c:j_c+n-1)$ with

$$\begin{align*}
\text{side} &= 'L' \\
\text{trans} &= 'N': Q \cdot \text{sub}(C) \\
&= 'H': Q^H \cdot \text{sub}(C)
\end{align*}$$

where $Q$ is a complex unitary distributed matrix of order $n_q$, with $n_q = m$ if $\text{side} = 'L'$ and $n_q = n$ if $\text{side} = 'R'$.

$Q$ is defined as the product of $ihi$-$ilo$ elementary reflectors, as returned by $\text{p?gehrd}$.

$Q = H(ilo) H(ilo+1) \ldots H(ihi-1)$.

Input Parameters

- **side** (global)
  - '='L': $Q$ or $Q^H$ is applied from the left.
  - '='R': $Q$ or $Q^H$ is applied from the right.

- **trans** (global)
  - '='N', no transpose, $Q$ is applied.
  - '='C', conjugate transpose, $Q^H$ is applied.

- **m** (global) The number of rows in the distributed matrix $\text{sub}(C)$ ($m \geq 0$).

- **n** (global) The number of columns in the distributed matrix $\text{sub}(C)$ ($n \geq 0$).

- **ilo, ihi** (global)
  - These must be the same parameters $ilo$ and $ihi$, respectively, as supplied to $\text{p?gehrd}$. $Q$ is equal to the unit matrix except in the distributed submatrix $Q(ia+ilo:ia+ihi-1,ja+ilo:ja+ihi-1)$.
  - If $\text{side} = 'L'$, then $1 \leq ilo \leq ihi \leq \text{max}(1,m)$.
  - If $\text{side} = 'R'$, then $1 \leq ilo \leq ihi \leq \text{max}(1,n)$

  $ilo$ and $ihi$ are relative indexes.

- **a** (local)
  - Pointer into the local memory to an array of size $lld_a*\text{LOCc}(ja+m-1)$ if $\text{side} = 'L'$, and $lld_a*\text{LOCc}(ja+n-1)$ if $\text{side} = 'R'$.
  - Contains the vectors which define the elementary reflectors, as returned by $\text{p?gehrd}$.

- **ia, ja** (global)
  - The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

- **desca** (global and local) array of size $\text{dlen}_\text{a}$. The array descriptor for the distributed matrix $A$.

- **tau** (local)
Array of size $\text{LOC}(ja+m-2)$, if $\text{side} = 'L'$, and $\text{LOC}(ja+n-2)$ if $\text{side} = 'R'$.

$\tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$ as returned by $\text{p?gehrd}$ ($0 \leq j < \text{size}(\tau)$). $\tau$ is tied to the distributed matrix $A$.

c
Pointer into the local memory to an array of size $\text{lld}_c \times \text{LOC}(jc+n-1)$.

Contains the local pieces of the distributed matrix $\text{sub}(C)$.

$\text{ic, jc}$

(global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix $C$, respectively.

$\text{descC}$

(global and local) array of size $\text{dlen}_c$. The array descriptor for the distributed matrix $C$.

$\text{work}$

(local)

Workspace array with size $lwork$.

$lwork$

(local or global)

The size of the array $\text{work}$.

$lwork$ must be at least $iaa = ia + ilo; jaa = ja + ilo - 1$;

If $\text{side} = 'L'$, $mi = ihi - ilo; ni = n; icc = ic + ilo; jcc = jc$;

$lwork \geq \max((\text{nb}_a*(\text{nb}_a-1))/2, (\text{numroc}(\text{numroc}(ni + icoffc, \text{nb}_a, 0, 0, \text{NPCOL}), \text{nb}_a, 0, 0, \text{lcmq})), \text{mpc0}) \times \text{nb}_a) + \text{nb}_a \times \text{nb}_a$

else if $\text{side} = 'R'$,

$mi = m; ni = ihi - ilo; icc = ic; jcc = jc + ilo; lwork \geq \max((\text{nb}_a*(\text{nb}_a-1))/2, (\text{numroc}(\text{numroc}(ni + icoffc, \text{nb}_a, 0, 0, \text{NPCOL}), \text{nb}_a, 0, 0, \text{lcmq})), \text{mpc0}) \times \text{nb}_a) + \text{nb}_a \times \text{nb}_a$

end if

where $\text{lcmq} = \text{lcm}/\text{NPCOL}$ with $\text{lcm} = \text{lcm}(\text{NPROW}, \text{NPCOL})$,

$\text{iroffa} = \text{mod}(iaa-1, \text{mb}_a),$

$\text{icoffa} = \text{mod}(jaa-1, \text{nb}_a),$

$\text{iarow} = \text{indxg2p}(iaa, \text{mb}_a, \text{MYROW}, \text{rsrca}, \text{NPROW}),$

$\text{npa0} = \text{numroc}(\text{numroc}(\text{numroc}(ni + icoffa, \text{mb}_a, \text{MYROW}, \text{iarow}, \text{NPROW}), \text{nb}_a, 0, 0, \text{lcmq})), \text{mpc0}) \times \text{nb}_a) + \text{nb}_a \times \text{nb}_a$

$\text{icrow} = \text{indxg2p}(icc, \text{mb}_c, \text{MYROW}, \text{rsrcc}, \text{NPROW}),$

$\text{iccol} = \text{indxg2p}(jcc, \text{nb}_c, \text{MYCOL}, \text{csrc_c}, \text{NPCOL}),$

$\text{mpc0} = \text{numroc}(\text{numroc}(\text{numroc}(mi + iroffc, \text{mb}_c, \text{MYROW}, \text{icrow}, \text{NPROW}), \text{nb}_a, 0, 0, \text{lcmq})), \text{mpc0}) \times \text{nb}_a) + \text{nb}_a \times \text{nb}_a$

$\text{mpc0} = \text{numroc}(\text{numroc}(\text{numroc}(mi + iroffc, \text{mb}_c, \text{MYROW}, \text{icrow}, \text{NPROW}), \text{nb}_a, 0, 0, \text{lcmq})), \text{mpc0}) \times \text{nb}_a) + \text{nb}_a \times \text{nb}_a$
NOTE
mod(x, y) is the integer remainder of x/y.

ilcm, indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

c
C is overwritten by Q* sub(C) or Q'*sub(C) or sub(C)*Q' or sub(C)*Q.

work[0])
On exit work[0] contains the minimum value of lwork required for optimum performance.

info (global)
= 0: the execution is successful.
< 0: if the i-th argument is an array and the j-th entry, indexed j - 1, had an illegal value, then info = -((i*100+j)); if the i-th argument is a scalar and had an illegal value, then info = -i.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lahqr
Computes the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form.

Syntax
void pslahqr (MKL_INT *wantt, MKL_INT *wantz, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, float *a, MKL_INT *desca, float *wr, float *wi, MKL_INT *iloz, MKL_INT *ihiz, float *z, MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *ilwork, MKL_INT *info );
void pdlahqr (MKL_INT *wantt, MKL_INT *wantz, MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, double *a, MKL_INT *desca, double *wr, double *wi, MKL_INT *iloz, MKL_INT *ihiz, double *z, MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *ilwork, MKL_INT *info );
void pclahqr (const MKL_INT *wantt, const MKL_INT *wantz, const MKL_INT *n, const MKL_INT *ilo, const MKL_INT *ihi, MKL_Complex8 *a, const MKL_INT *desca, MKL_Complex8 *w, const MKL_INT *iloz, const MKL_INT *ihiz, MKL_Complex8 *z, const MKL_INT *descz, MKL_Complex8 *work, const MKL_INT *lwork, const MKL_INT *iwork, const MKL_INT *ilwork, MKL_INT *info );
void pzlahqr (const MKL_INT *wantt, const MKL_INT *wantz, const MKL_INT *n, const MKL_INT *ilo, const MKL_INT *ihi, MKL_Complex16 *a, const MKL_INT *desca, MKL_Complex16 *w, const MKL_INT *iloz, const MKL_INT *ihiz, MKL_Complex16 *z, const MKL_INT *descz, MKL_Complex16 *work, const MKL_INT *lwork, const MKL_INT *iwork, const MKL_INT *ilwork, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

This is an auxiliary function used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns ilo and ihi.

NOTE

These restrictions apply to the use of p?lahqr:

• The code requires the distributed block size to be square and at least 6.
• The code requires A and Z to be distributed identically and have identical contexts.
• The matrix A must be in upper Hessenberg form. If elements below the subdiagonal are non-zero, the resulting transformations can be nonsimilar.
• All eigenvalues are distributed to all the nodes.

Input Parameters

wantt (global)
If wantt≠ 0, the full Schur form T is required;
If wantt = 0, only eigenvalues are required.

wantz (global)
If wantz≠ 0, the matrix of Schur vectors Z is required;
If wantz = 0, Schur vectors are not required.

n (global) The order of the Hessenberg matrix A (and z if wantz is non-zero). n≥0.

ило, ihi (global)
It is assumed that A is already upper quasi-triangular in rows and columns ihi+1:n, and that A(ilo, ilo-1) = 0 (unless ilo = 1). p?lahqr works primarily with the Hessenberg submatrix in rows and columns ilo to ihi, but applies transformations to all of H if wantt is non-zero. 1≤ilo≤max(1,ihi); ihi ≤ n.

a (global)
Array, of size lld_a * LOCc(n). On entry, the upper Hessenberg matrix A.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

iloz, ihiz (global) Specify the rows of the matrix Z to which transformations must be applied if wantz is non-zero. 1≤iloz≤ilo; ihi≤ihiz≤n.
Array. If wantz is non-zero, on entry z must contain the current matrix Z of transformations accumulated by pdhseqr. If wantz is zero, z is not referenced.

descz
(global and local) array of size dlen_. The array descriptor for the distributed matrix Z.

work
(local)
Workspace array with size lwork.

lwork
(local) The size of work. lwork is assumed big enough so that lwork≥3*n + max(2*max(lld_z, lld_a) + 2*LOCq(n), 7*ceil(n/hbl)/lcm(NPROW, NPCOL))).

If lwork = -1, then work[0] gets set to the above number and the code returns immediately.

iwork
(global and local) array of size ilwork. Not referenced and can be NULL pointer.

ilwork
(local) This holds some of the iblk integer arrays. Not referenced and can be NULL pointer.

Output Parameters

a
On exit, if wantt is non-zero, A is upper quasi-triangular in rows and columns ilo:ihi, with any 2-by-2 or larger diagonal blocks not yet in standard form. If wantt is zero, the contents of A are unspecified on exit.

work[0]
On exit work[0] contains the minimum value of lwork required for optimum performance.

wr, wi
(global replicated output)
Arrays of size n each. The real and imaginary parts, respectively, of the computed eigenvalues ilo to ihi are stored in the corresponding elements of wr and wi. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi, say the i-th and (i+1)-th, with wi[i-1]> 0 and wi[i] < 0. If wantt is zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in A. A may be returned with larger diagonal blocks until the next release.

w
(global replicated output)
Array of size n. The computed eigenvalues ilo to ihi are stored in the corresponding elements of w. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of w, say the i-th and (i+1)-th, with w[i-1]> 0 and w[i] < 0. If wantt is zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in A. A may be returned with larger diagonal blocks until the next release.

z
On exit z has been updated; transformations are applied only to the submatrix Z(ilo:ihi, ilo:ihi).

info
(global)
= 0: the execution is successful.
< 0: the parameter number - info is incorrect or inconsistent
> 0: p?lahqr failed to compute all the eigenvalues ilo to ihi in a total of
30*(ihi-ilo+1) iterations; if info = i, elements i+1 to ihi of wr and wi
contain the eigenvalues that have been successfully computed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?trevc
Computes right and/or left eigenvectors of a complex upper triangular matrix in parallel.

Syntax
void pctrevc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, MKL_Complex8* t, const MKL_INT* desct, MKL_Complex8* vl, const MKL_INT*
descvl, MKL_Complex8* vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m,
MKL_Complex8* work, float* rwork, MKL_INT* info);
void pztrevc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, MKL_Complex16* t, const MKL_INT* desct, MKL_Complex16* vl, const MKL_INT*
descvl, MKL_Complex16* vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m,
MKL_Complex16* work, double* rwork, MKL_INT* info);
void pdtrevc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, double* t, const MKL_INT* desct, double* vl, const MKL_INT* descvl, double*
vr, const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m, double* work, MKL_INT* info);
void pstrevc (const char* side, const char* howmny, const MKL_INT* select, const
MKL_INT* n, float* t, const MKL_INT* desct, float* vl, const MKL_INT* descvl, float* vr,
const MKL_INT* descvr, const MKL_INT* mm, MKL_INT* m, float* work, MKL_INT* info);

Include Files
• mkl_scalapack.h

Description
p?trevc computes some or all of the right and/or left eigenvectors of a complex upper triangular matrix T in
parallel.

The right eigenvector x and the left eigenvector y of T corresponding to an eigenvalue w are defined by:
T*x = w*x,
y'*T = w*y'
where y' denotes the conjugate transpose of the vector y.
If all eigenvectors are requested, the routine may either return the matrices X and/or Y of right or left
eigenvectors of T, or the products Q*X and/or Q*Y, where Q is an input unitary matrix. If T was obtained
from the Schur factorization of an original matrix A = Q*T*Q', then Q*X and Q*Y are the matrices of right or
left eigenvectors of A.

Input Parameters

side (global)
howmny
(global)
= 'A': compute all right and/or left eigenvectors;
= 'B': compute all right and/or left eigenvectors, and backtransform them
using the input matrices supplied in vr and/or vl;
= 'S': compute selected right and/or left eigenvectors, specified by the
logical array select.

select
(global)
Array, size (n)
If howmny = 'S', select specifies the eigenvectors to be computed.
If howmny = 'A' or 'B', select is not referenced. To select the eigenvector
 corresponding to the j-th eigenvalue, select[j - 1] must be set to non-zero.

n
(global)
The order of the matrix T. n >= 0.

t
(local)
Array, size lld_t*LOCc(n).
The upper triangular matrix T. T is modified, but restored on exit.

desct
(global and local)
Array of size dlen_.
The array descriptor for the distributed matrix T.

vl
(local)
Array, size (descvl(lld_), mm)
On entry, if side = 'L' or 'B' and howmny = 'B', vl must contain an n-by-n
matrix Q (usually the unitary matrix Q of Schur vectors returned
by ?hseqr).

descvl
(global and local)
Array of size dlen_.
The array descriptor for the distributed matrix VL.

vr
(local)
Array, size descvr(lld_)*mm.
On entry, if side = 'R' or 'B' and howmny = 'B', vr must contain an n-by-n
matrix Q (usually the unitary matrix Q of Schur vectors returned
by ?hseqr).

descvr
(global and local)
Array of size dlen_.

= 'R': compute right eigenvectors only;
= 'L': compute left eigenvectors only;
= 'B': compute both right and left eigenvectors.
The array descriptor for the distributed matrix \( VR \).

\( mm \) (global)

The number of columns in the arrays \( vl \) and/or \( vr \). \( mm \geq m \).

\( work \) (local)

Array, size \( \left( 2 \times \text{desct}(lld_\_\_ ) \right) \)

Additional workspace may be required if \( p\text{lattrs} \) is updated to use \( work \).

\( rwork \) (local)

Array, size \( \left( \text{desct}(lld_\_\_ ) \right) \)

**Output Parameters**

\( t \)

The upper triangular matrix \( T \). \( T \) is modified, but restored on exit.

\( vl \)

On exit, if \( side = 'L' \) or \( 'B' \), \( vl \) contains:

- If \( howmny = 'A' \), the matrix \( Y \) of left eigenvectors of \( T \);
- If \( howmny = 'B' \), the matrix \( Q^*Y \);
- If \( howmny = 'S' \), the left eigenvectors of \( T \) specified by \( select \), stored consecutively in the columns of \( vl \), in the same order as their eigenvalues. If \( side = 'R' \), \( vl \) is not referenced.

\( vr \)

On exit, if \( side = 'R' \) or \( 'B' \), \( vr \) contains:

- If \( howmny = 'A' \), the matrix \( X \) of right eigenvectors of \( T \);
- If \( howmny = 'B' \), the matrix \( Q^*X \);
- If \( howmny = 'S' \), the right eigenvectors of \( T \) specified by \( select \), stored consecutively in the columns of \( vr \), in the same order as their eigenvalues. If \( side = 'L' \), \( vr \) is not referenced.

\( m \) (global)

The number of columns in the arrays \( vl \) and/or \( vr \) actually used to store the eigenvectors. If \( howmny = 'A' \) or \( 'B' \), \( m \) is set to \( n \). Each selected eigenvector occupies one column.

\( info \) (global)

- \( = 0 \): successful exit
- \( < 0 \): if \( info = -i \), the i-th argument had an illegal value

**Application Notes**

The algorithm used in this program is basically backward (forward) substitution. Scaling should be used to make the code robust against possible overflow. But scaling has not yet been implemented in \( p\text{lattrs} \) which is called by this routine to solve the triangular systems. \( p\text{lattrs} \) just calls \( p\text{trsv} \).

Each eigenvector is normalized so that the element of largest magnitude has magnitude 1; here the magnitude of a complex number \((x,y)\) is taken to be \( |x| + |y| \).

**Singular Value Decomposition: ScaLAPACK Driver Routines**

This section describes ScaLAPACK routines for computing the singular value decomposition (SVD) of a general \( m \)-by-\( n \) matrix \( A \) (see "Singular Value Decomposition" in LAPACK chapter).
To find the SVD of a general matrix $A$, this matrix is first reduced to a bidiagonal matrix $B$ by a unitary (orthogonal) transformation, and then SVD of the bidiagonal matrix is computed. Note that the SVD of $B$ is computed using the LAPACK routine `?bdsqr`.

Table "Computational Routines for Singular Value Decomposition (SVD)" lists ScaLAPACK computational routines for performing this decomposition.

### Computational Routines for Singular Value Decomposition (SVD)

<table>
<thead>
<tr>
<th>Operation</th>
<th>General matrix</th>
<th>Orthogonal/unitary matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce $A$ to a bidiagonal matrix</td>
<td>p?gebrd</td>
<td>p?ormbr/p?unmbr</td>
</tr>
<tr>
<td>Multiply matrix after reduction</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### p?gebrd

*Reduces a general matrix to bidiagonal form.*

**Syntax**

```c
void psgebrd (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, float *tauq, float *taup, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgebrd (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, double *tauq, double *taup, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgebrd (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, MKL_Complex8 *tauq, MKL_Complex8 *taup, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgebrd (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, MKL_Complex16 *tauq, MKL_Complex16 *taup, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?gebrd function reduces a real/complex general $m$-by-$n$ distributed matrix $\text{sub}(A) = A(\text{i}a:\text{i}a+m-1, j\text{a}:j\text{a}+n-1)$ to upper or lower bidiagonal form $B$ by an orthogonal/unitary transformation:

$$Q^*\text{sub}(A)*P = B.$$ 

If $m \geq n$, $B$ is upper bidiagonal; if $m < n$, $B$ is lower bidiagonal.

**Input Parameters**

- $m$ (global) The number of rows in the distributed matrix $\text{sub}(A)$ ($m \geq 0$).
- $n$ (global) The number of columns in the distributed matrix $\text{sub}(A)$ ($n \geq 0$).
- $a$ (local)
  
  Real pointer into the local memory to an array of size $\text{lld}_a*\text{LOCc}(j\text{a}+n-1)$.

  On entry, this array contains the distributed matrix sub $(A)$.
- $\text{i}a, j\text{a}$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.
desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

work (local) Workspace array of size lwork.

lwork (local or global) size of work, must be at least:
\[ \text{lwork} \geq \text{nb} \times (\text{mpa} + \text{nqa} + 1) + \text{nqa} \]
where \( \text{nb} = \text{mb}_a = \text{nb}_a \),
\[ \text{iroffa} = \text{mod}(\text{ia}-1, \text{nb}), \]
\[ \text{icoffa} = \text{mod}(\text{ja}-1, \text{nb}), \]
\[ \text{iarow} = \text{indxg2p}(\text{ia}, \text{nb}, \text{MYROW}, \text{rsrc}_a, \text{NPROW}), \]
\[ \text{iacol} = \text{indxg2p}(\text{ja}, \text{nb}, \text{MYCOL}, \text{csrc}_a, \text{NPCOL}), \]
\[ \text{mpa} = \text{numroc}(\text{m} + \text{iroffa}, \text{nb}, \text{MYROW}, \text{iarow}, \text{NPROW}), \]
\[ \text{nqa} = \text{numroc}(\text{n} + \text{icoffa}, \text{nb}, \text{MYCOL}, \text{iacol}, \text{NPCOL}), \]

**NOTE**
\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

indxg2p and numroc are ScALAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If \( \text{lwork} = -1 \), then \( \text{lwork} \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

**Output Parameters**

\( a \)
On exit, if \( m \geq n \), the diagonal and the first superdiagonal of sub(\( A \)) are overwritten with the upper bidiagonal matrix \( B \); the elements below the diagonal, with the array taug, represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors, and the elements above the first superdiagonal, with the array taup, represent the orthogonal matrix \( P \) as a product of elementary reflectors. If \( m < n \), the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix \( B \); the elements below the first subdiagonal, with the array taug, represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors, and the elements above the diagonal, with the array taup, represent the orthogonal matrix \( P \) as a product of elementary reflectors. See Application Notes below.

\( d \)
Array of size \( \text{LOCc}(ja+\min(m,n)-1) \) if \( m \geq n \) and \( \text{LOCr}(ia+\min(m,n)-1) \) otherwise. The distributed diagonal elements of the bidiagonal matrix \( B \):
\[ d[i] = A(i+1,i+1), \quad 0 \leq i < \text{size}(d). \]
\( d \) is tied to the distributed matrix \( A \).

\( e \)
(local)
Array of size $\text{LOCr}(i_a+\min(m,n)-1)$ if $m \geq n$; $\text{LOCc}(j_a+\min(m,n)-2)$ otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix $B$:

If $m \geq n$, $e[i] = A(i+1,i+2)$ for $i = 0,1,\ldots, n-2$; if $m < n$, $e[i] = A(i+2,i+1)$ for $i = 0,1,\ldots, m-2$. $e$ is tied to the distributed matrix $A$.

$\tau_q, \tau_p$

(local)

Arrays of size $\text{LOCc}(j_a+\min(m,n)-1)$ for $\tau_q$ and $\text{LOCr}(i_a+\min(m,n)-1)$ for $\tau_p$. Contain the scalar factors of the elementary reflectors that represent the orthogonal/unitary matrices $Q$ and $P$, respectively. $\tau_q$ and $\tau_p$ are tied to the distributed matrix $A$. See Application Notes below.

work[0]

On exit work[0] contains the minimum value of lwork required for optimum performance.

info

(global)

$= 0$: the execution is successful.

$< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $\text{info} = -(\text{INFO}+j)$; if the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

The matrices $Q$ and $P$ are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1)*H(2)*\ldots*H(n)$, and $P = G(1)*G(2)*\ldots*G(n-1)$.

Each $H(i)$ and $G(i)$ has the form:

$H(i) = i - \tau_q * v * v'$ and $G(i) = i - \tau_p * u * u'$

where $\tau_q$ and $\tau_p$ are real/complex scalars, and $v$ and $u$ are real/complex vectors;

$v(1:i-1) = 0$, $v(i) = 1$, and $v(i+1:m)$ is stored on exit in $A(i_a+i:i_a+m-1,j_a+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(i_a+i-1,j_a+i+1:j_a+n-1)$;

$\tau_q$ is stored in $\tau_q[a+i-2]$ and $\tau_p$ in $\tau_p[a+i-2]$.

If $m < n$,

$Q = H(1)*H(2)*\ldots*H(m-1)$, and $P = G(1)*G(2)*\ldots*G(m)$

Each $H(i)$ and $G(i)$ has the form:

$H(i) = i - \tau_q * v * v'$ and $G(i) = i - \tau_p * u * u'$

here $\tau_q$ and $\tau_p$ are real/complex scalars, and $v$ and $u$ are real/complex vectors;

$v(1:i) = 0$, $v(i+1) = 1$, and $v(i+2:m)$ is stored on exit in $A(i_a+i:i_a+m-1,j_a+i-1)$;

$u(1:i-1) = 0$, $u(i) = 1$, and $u(i+1:n)$ is stored on exit in $A(i_a+i-1,j_a+i+1:j_a+n-1)$;

$\tau_q$ is stored in $\tau_q[a+i-2]$ and $\tau_p$ in $\tau_p[a+i-2]$.

The contents of sub($A$) on exit are illustrated by the following examples:

$m = 6$ and $n = 5 (m > n)$:
where $d$ and $e$ denote diagonal and off-diagonal elements of $B$, $v_i$ denotes an element of the vector defining $H(i)$, and $u_i$ an element of the vector defining $G(i)$.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ormbr**

Multiplies a general matrix by one of the orthogonal matrices from a reduction to bidiagonal form determined by p?gebrd.

**Syntax**

```c
void psormbr (char *vect, char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pdormbr (char *vect, char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

If vect = 'Q', the p?ormbr function overwrites the general real distributed $m$-by-$n$ matrix sub($C$) = $C(\text{ic}:	ext{ic} + m-1, \text{jc}:	ext{jc} + n-1)$ with

$$side = 'L' \quad \quad side = 'R'$$
### Input Parameters

**vect** (global)
- If `vect = 'Q'`, then $Q$ or $Q^T$ is applied.
- If `vect = 'P'`, then $P$ or $P^T$ is applied.

**side** (global)
- If `side = 'L'`, then $Q$ or $Q^T$, $P$ or $P^T$ is applied from the left.
- If `side = 'R'`, then $Q$ or $Q^T$, $P$ or $P^T$ is applied from the right.

**trans** (global)
- If `trans = 'N'`, no transpose, $Q$ or $P$ is applied.
- If `trans = 'T'`, then $Q^T$ or $P^T$ is applied.

**m** (global) The number of rows in the distributed matrix sub(C).

**n** (global) The number of columns in the distributed matrix sub(C).

**k** (global)
- If `vect = 'Q'`, the number of columns in the original distributed matrix reduced by `pgebrd`.
- If `vect = 'P'`, the number of rows in the original distributed matrix reduced by `pgebrd`.

**Constraints:** $k \geq 0$. 

**a** (local)
Pointer into the local memory to an array of size lld_a * LOCC(ja + min(nq,k)-1) if vect='Q', and lld_a * LOCC(ja+nq-1) if vect = 'P'.

nq = m if side = 'L', and nq = n otherwise.

The vectors that define the elementary reflectors H(i) and G(i), whose products determine the matrices Q and P, as returned by p?gebrd.

If vect = 'Q', lld_a≥max(1, LOCr(ia+nq-1));
If vect = 'P', lld_a≥max(1, LOCr(ia+min(nq, k)-1)).

ia, ja (global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

tau (local)
Array of size LOCC(ja+min(nq, k)-1), if vect = 'Q', and LOCr(ia +min(nq, k)-1), if vect = 'P'.

tau[i] must contain the scalar factor of the elementary reflector H(i+1) or G(i+1)
which determines Q or P, as returned by pdgebrd in its array argument tauq or taup. tau is tied to the distributed matrix A.

c (local)
Pointer into the local memory to an array of size lld_c*LOCC(jc+n-1).
Contains the local pieces of the distributed matrix sub (C).

ic, jc (global) The row and column indices in the global matrix C indicating the first row and the first column of the submatrix C, respectively.

desc c (global and local) array of size dlen_. The array descriptor for the distributed matrix C.

work (local)
Workspace array of size lwork.

lwork (local or global) size of work, must be at least:

If side = 'L'
  nq = m;
  if ((vect = 'Q' and nq≥k) or (vect is not equal to 'Q' and nq>k)),
  iaa=ia; jaa=ja; mi=m; ni=n; icc=ic; jcc=jc;
else
  iaa= ia+1; jaa=ja; mi=m-1; ni=n; icc=ic+1; jcc= jc;
end if
else
If side = 'R', nq = n;
if((vect = 'Q' and nq≥k) or (vect is not equal to 'Q' and nq>k)),

\( i_{aa} = i_a; \ j_{aa} = j_a; \ m_i = m; \ n_i = n; \ i_{cc} = i_c; \ j_{cc} = j_c; \)

else

\( i_{aa} = i_a; \ j_{aa} = j_a + 1; \ m_i = m; \ n_i = n - 1; \ i_{cc} = i_c; \ j_{cc} = j_c + 1; \)

end if

end if

If \( \text{vect} = 'Q' \),

If \( \text{side} = 'L' \), \( l_{work} \geq \max((n_b_a \times (n_b_a - 1))/2, (n_q_c_0 + m_p_c_0) \times n_b_a) + n_b_a \times n_b_a \)

else if \( \text{side} = 'R' \),

\( l_{work} \geq \max((n_b_a \times (n_b_a - 1))/2, (n_q_c_0 + n_p_a_0) + \text{numroc}(\text{numroc}(n_i + i_{coff_a}, n_b_a, 0, 0, \text{NPCOL}), n_b_a, 0, 0, l_{cm}, m_p_c_0) \times n_b_a) + n_b_a \times n_b_a \)

end if

else if \( \text{vect} \) is not equal to \( 'Q' \), if \( \text{side} = 'L' \),

\( l_{work} \geq \max((m_b_a \times (m_b_a - 1))/2, (m_p_c_0 + \max(m_q_a_0 + \text{numroc}(\text{numroc}(m_i + i_{roff_c}, m_b_a, 0, 0, \text{NPROW}), m_b_a, 0, 0, l_{cmp}), n_q_c_0) \times m_b_a) + m_b_a \times m_b_a \)

else if \( \text{side} = 'R' \),

\( l_{work} \geq \max((m_b_a \times (m_b_a - 1))/2, (m_p_c_0 + n_q_c_0) \times m_b_a) + m_b_a \times m_b_a \)

end if

end if

where \( l_{cmp} = l_{cm}/\text{NPROW}, \ l_{cmq} = l_{cm}/\text{NPCOL} \), with \( l_{cm} = \text{lcm}(\text{NPROW}, \ \text{NPCOL}) \),

\( i_{roff_a} = \text{mod}(i_{aa}-1, m_b_a) \),

\( i_{coff_a} = \text{mod}(j_{aa}-1, n_b_a) \),

\( i_{arow} = \text{indxg2p}(i_{aa}, m_b_a, \ \text{MYROW}, \ rsrc_a, \ \text{NPROW}) \),

\( i_{icol} = \text{indxg2p}(j_{aa}, n_b_a, \ \text{MYCOL}, \ csrc_a, \ \text{NPCOL}) \),

\( m_q_a_0 = \text{numroc}(n_i + i_{coff_a}, n_b_a, \ \text{MYCOL}, \ i_{arow}, \ \text{NPROW}) \),

\( n_p_a_0 = \text{numroc}(n_i + i_{roff_a}, m_b_a, \ \text{MYROW}, \ i_{arlo}, \ \text{NPROW}) \),

\( i_{roff_c} = \text{mod}(i_{cc}-1, m_b_c) \),

\( i_{coff_c} = \text{mod}(j_{cc}-1, n_b_c) \),

\( i_{icrow} = \text{indxg2p}(i_{cc}, m_b_c, \ \text{MYROW}, \ rsrc_c, \ \text{NPROW}) \),

\( i_{icol} = \text{indxg2p}(j_{cc}, n_b_c, \ \text{MYCOL}, \ csrc_c, \ \text{NPCOL}) \),

\( m_p_c_0 = \text{numroc}(m_i + i_{roff_c}, m_b_c, \ \text{MYROW}, \ i_{icrow}, \ \text{NPROW}) \),

\( n_q_c_0 = \text{numroc}(n_i + i_{coff_c}, n_b_c, \ \text{MYCOL}, \ i_{icol}, \ \text{NPCOL}) \),

**NOTE**

\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).
indxg2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

\text{c}

On exit, if vect='Q', sub(C) is overwritten by Q*sub(C), or Q'*sub(C), or sub(C)*Q', or sub(C)*Q; if vect='P', sub(C) is overwritten by P*sub(C), or P'*sub(C), or sub(C)*P, or sub(C)*P'.

work[0]

On exit work[0] contains the minimum value of lwork required for optimum performance.

\text{info}

(global)

\text{info} = 0: the execution is successful.

\text{info} < 0: if the \text{i}-th argument is an array and the \text{j}-th entry, indexed \text{j} - 1, had an illegal value, then \text{info} = -(\text{i}*100+\text{j}); if the \text{i}-th argument is a scalar and had an illegal value, then \text{info} = -\text{i}.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\text{p?unmbr}

Multiplies a general matrix by one of the unitary transformation matrices from a reduction to bidiagonal form determined by \text{p?gebrd}.

Syntax

\begin{verbatim}
void pcunmbr (char *vect, char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzunmbr (char *vect, char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
\end{verbatim}

Include Files

- mkl_scalapack.h

Description

If vect = 'Q', the \text{p?unmbr} function overwrites the general complex distributed \text{m}-by-\text{n} matrix sub(C) = C(\text{ic}:\text{ic}+\text{m}-1,\text{jc}:\text{jc}+\text{n}-1) with

\begin{align*}
\text{trans} = 'N': & \quad \text{Q*sub(C)} & \quad \text{trans} = 'C': & \quad \text{Q}^\dagger*\text{sub(C)} \\
\text{trans} = 'L': & \quad \text{Q} & \quad \text{trans} = 'R': & \quad \text{Q}^\dagger
\end{align*}
If vect = 'P', the function overwrites sub(C) with

\[ \begin{align*}
\text{side} &= \text{'L'} & \text{side} &= \text{'R'} \\
\text{trans} &= \text{'N'}: & P^{*}\text{sub}(C) & & \text{sub}(C)^{*}P \\
\text{trans} &= \text{'C'}: & P^{H}\text{sub}(C) & & \text{sub}(C)^{*}P^{H}
\end{align*} \]

Here Q and PH are the unitary distributed matrices determined by p?gebrd when reducing a complex distributed matrix \( A(ia:* , ja:* ) \) to bidiagonal form: \( A(ia:* , ja:* ) = Q^{*}B^{*}P^{H} \).

Q and PH are defined as products of elementary reflectors \( H(i) \) and \( G(i) \) respectively.

Let \( nq = m \) if side = 'L' and \( nq = n \) if side = 'R'. Therefore \( nq \) is the order of the unitary matrix \( Q \) or \( P^{H} \) that is applied.

If vect = 'Q', \( A(ia:* , ja:* ) \) is assumed to have been an \( nq \)-by-\( k \) matrix:

1. If \( nq \geq k \), \( Q = H(1)H(2)\ldots H(k) \);
2. If \( nq < k \), \( Q = H(1)H(2)\ldots H(nq-1) \).

If vect = 'P', \( A(ia:* , ja:* ) \) is assumed to have been a \( k \)-by-\( nq \) matrix:

1. If \( k < nq \), \( P = G(1)G(2)\ldots G(k) \);
2. If \( k \geq nq \), \( P = G(1)G(2)\ldots G(nq-1) \).

**Input Parameters**

**vect**

(global)

- If vect = 'Q', then Q or \( Q^{H} \) is applied.
- If vect = 'P', then P or \( P^{H} \) is applied.

**side**

(global)

- If side = 'L', then Q or \( Q^{H} \), P or \( P^{H} \) is applied from the left.
- If side = 'R', then Q or \( Q^{H} \), P or \( P^{H} \) is applied from the right.

**trans**

(global)

- If trans = 'N', no transpose, Q or P is applied.
- If trans = 'C', conjugate transpose, \( Q^{H} \) or \( P^{H} \) is applied.

**m**

(global) The number of rows in the distributed matrix sub (C) \( m \geq 0 \).

**n**

(global) The number of columns in the distributed matrix sub (C) \( n \geq 0 \).

**k**

(global)

- If vect = 'Q', the number of columns in the original distributed matrix reduced by p?gebrd;
- If vect = 'P', the number of rows in the original distributed matrix reduced by p?gebrd.

Constraints: \( k \geq 0 \).

**a**

(local)

Pointer into the local memory to an array of size \( lld_{a} \times \text{LOCc}(ja + \min(nq,k)-1) \) if vect = 'Q', and \( lld_{a} \times \text{LOCc}(ja+nq-1) \) if vect = 'P'.

- \( nq = m \) if side = 'L', and \( nq = n \) otherwise.
The vectors that define the elementary reflectors \( H(i) \) and \( G(i) \), whose products determine the matrices \( Q \) and \( P \), as returned by \texttt{pgebrd}.

If \( \text{vect} = 'Q' \), \( \text{lld}_a \geq \max(1, \text{LOCr}(i\text{a} + nq - 1)) \);
If \( \text{vect} = 'P' \), \( \text{lld}_a \geq \max(1, \text{LOCr}(i\text{a} + \min(nq, k) - 1)) \).

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \text{tau} \) (local)
Array of size \( \text{LOCc}(ja + \min(nq, k) - 1) \), if \( \text{vect} = 'Q' \), and \( \text{LOCr}(ia + \min(nq, k) - 1) \), if \( \text{vect} = 'P' \).
\( \text{tau}[i] \) must contain the scalar factor of the elementary reflector \( H(i+1) \) or \( G(i+1) \), which determines \( Q \) or \( P \), as returned by \texttt{pgebrd} in its array argument \text{taqu} or \text{taup}. \( \text{tau} \) is tied to the distributed matrix \( A \).

\( c \) (local)
Pointer into the local memory to an array of size \( lld_c*\text{LOCc}(jc+n-1) \).
Contains the local pieces of the distributed matrix sub \( (C) \).

\( ic, jc \) (global) The row and column indices in the global matrix \( C \) indicating the first row and the first column of the submatrix \( C \), respectively.

\( \text{descC} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( C \).

\( \text{work} \) (local)
Workspace array of size \( lwork \).

\( lwork \) (local or global) size of \( \text{work} \), must be at least:
If \( \text{side} = 'L' \)
\( nq = m; \)
if \( ((\text{vect} = 'Q' \text{ and } nq \geq k) \text{ or (vect is not equal to 'Q' and } nq>k)) \),
\( iaa = ia; jaa = ja; mi = m; ni = n; icc = ic; jcc = jc; \)
else
\( iaa = ia+1; jaa = ja; mi = m-1; ni = n; icc = ic+1; jcc = jc; \)
end if
else
If \( \text{side} = 'R', nq = n; \)
if \( ((\text{vect} = 'Q' \text{ and } nq \geq k) \text{ or (vect is not equal to 'Q' and } nq>k)) \),
\( iaa = ia; jaa = ja; mi = m; ni = n; icc = ic; jcc = jc; \)
else
\( iaa = ia; jaa = ja+1; mi = m; ni = n-1; icc = ic; jcc = jc+1; \)
end if
end if

If vect = 'Q',
If side = 'L', lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0+mpc0)*nb_a) + nb_a*nb_a
else if side = 'R',
lwork ≥ max((nb_a*(nb_a-1))/2, (nqc0 + max(npa0+numroc(numroc(mi+icoffc, nb_a, 0, 0, NPCOL), nb_a, 0, 0, lcmp), mpc0))*nb_a) + nb_a*nb_a
end if
else if vect is not equal to 'Q',
if side = 'L',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + max(mqa0+numroc(numroc(mi+icoffc, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcmp), nqc0))*mb_a) + mb_a*mb_a
else if side = 'R',
lwork ≥ max((mb_a*(mb_a-1))/2, (mpc0 + nqc0)*mb_a) + mb_a*mb_a
end if
end if

where lcmp = lcm/NPROW, lcmq = lcm/NPCOL, with lcm = ilcm(NPROW, NPCOL),
iraff0 = mod(iaa-1, mb_a),
icozza = mod(jaa-1, nb_a),
 iarow = indxg2p(iaa, mb_a, MYROW, rsrca, NPROW),
iacol = indxg2p(jaa, nb_a, MYCOL, csrca, NPCOL),
mqa0 = numroc(mi+icoffic, nb_a, MYCOL, iacol, NPCOL),
npa0 = numroc(mi+icoffic, mb_a, MYROW, iarow, NPROW),
 iroffce = mod(icc-1, mb_c),
icoffe = mod(jcc-1, nb_c),
icrow = indxg2p(icc, mb_c, MYROW, rsrcc, NPROW),
iccol = indxg2p(jcc, nb_c, MYCOL, csrcc, NPCOL),
mpc0 = numroc(mi+iroffcc, mb_c, MYROW, icrow, NPROW),
nqc0 = numroc(mi+icoffic, nb_c, MYCOL, iccol, NPCOL),

NOTE
mod(x,y) is the integer remainder of x/y.
indxg2p and numroc are ScALAPACK tool functions; MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

$c$

On exit, if vect='Q', sub(C) is overwritten by $Q^{*}\text{sub}(C)$, or $Q^{*}\text{sub}(C)$, or sub(C)*$Q'$, or sub(C)*$Q'$; if vect='P', sub(C) is overwritten by $P^{*}\text{sub}(C)$, or $P^{*}\text{sub}(C)$, or sub(C)*$P$, or sub(C)*$P$.

$work[0]$

On exit $work[0]$ contains the minimum value of $lwork$ required for optimum performance.

$info$

(global)

= 0: the execution is successful.

< 0: if the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also

Overview for details of ScALAPACK array descriptor structures and related notations.

Generalized Symmetric-Definite Eigenvalue Problems: ScALAPACK Computational Routines

This section describes ScALAPACK routines that allow you to reduce the generalized symmetric-definite eigenvalue problems (see Generalized Symmetric-Definite Eigenvalue Problems in LAPACK chapters) to standard symmetric eigenvalue problem $Cy = \lambda y$, which you can solve by calling ScALAPACK routines described earlier in this chapter (see Symmetric Eigenproblems).

Table "Computational Routines for Reducing Generalized Eigenproblems to Standard Problems" lists these routines.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Real symmetric matrices</th>
<th>Complex Hermitian matrices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reduce to standard problems</td>
<td>p?sygst</td>
<td>p?hegst</td>
</tr>
</tbody>
</table>

$p?sygst$

Reduces a real symmetric-definite generalized eigenvalue problem to the standard form.

Syntax

void pssygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *scale , MKL_INT *info );

void pdsygst (MKL_INT *ibtype , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *scale , MKL_INT *info );
Include Files

- mkl_scalapack.h

Description

The `p?sygst` function reduces real symmetric-definite generalized eigenproblems to the standard form.

In the following sub\((A)\) denotes \(A(ia:ia+n-1, ja:ja+n-1)\) and sub\((B)\) denotes \(B(ib:ib+n-1, jb:jb+n-1)\).

If \(ibtype = 1\), the problem is

\[ sub(A)^*x = \lambda*sub(B)^*x, \]

and \(sub(A)\) is overwritten by \(\text{inv}(U^T)*sub(A)^*\text{inv}(U)\), or \(\text{inv}(L)*sub(A)^*\text{inv}(L^T)\).

If \(ibtype = 2\) or \(3\), the problem is

\[ sub(A)^*sub(B)^*x = \lambda*x, \]
\[ \text{or} \quad sub(B)^*sub(A)^*x = \lambda*x, \]

and \(sub(A)\) is overwritten by \(U*sub(A)^*U^T\), or \(L^T*sub(A)^*L\).

\(sub(B)\) must have been previously factorized as \(U^T*U\) or \(L*L^T\) by \(p?potrf\).

Input Parameters

\(ibtype\)  
(global) Must be 1 or 2 or 3.
- If \(itype = 1\), compute \(\text{inv}(U^T)*sub(A)^*\text{inv}(U)\), or \(\text{inv}(L)*sub(A)^*\text{inv}(L^T)\);
- If \(itype = 2\) or \(3\), compute \(U*sub(A)^*U^T\), or \(L^T*sub(A)^*L\).

\(uplo\)  
(global) Must be '\(U'\) or '\(L'\).
- If \(uplo = 'U'\), the upper triangle of \(sub(A)\) is stored and \(sub(B)\) is factored as \(U^T*U\).
- If \(uplo = 'L'\), the lower triangle of \(sub(A)\) is stored and \(sub(B)\) is factored as \(L*L^T\).

\(n\)  
(global) The order of the matrices \(sub(A)\) and \(sub(B)\) \((n \geq 0)\).

\(a\)  
(local)
Pointer into the local memory to an array of size \(lld_a*LOCc(ja+n-1)\). On entry, the array contains the local pieces of the \(n\)-by-\(n\) symmetric distributed matrix \(sub(A)\).
- If \(uplo = 'U'\), the leading \(n\)-by-\(n\) upper triangular part of \(sub(A)\) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.
- If \(uplo = 'L'\), the leading \(n\)-by-\(n\) lower triangular part of \(sub(A)\) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

\(ia, ja\)  
(global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.

\(desca\)  
(global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(A\).

\(b\)  
(local)
Pointer into the local memory to an array of size $ld_b*LOCc(jb+n-1)$. On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of sub $(B)$ as returned by `p?potrf`.

$ib, jb$  
(global) The row and column indices in the global matrix $B$ indicating the first row and the first column of the submatrix $B$, respectively.

descb  
(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $B$.

**Output Parameters**

$a$  
On exit, if $info = 0$, the transformed matrix, stored in the same format as sub$(A)$.

$scale$  
(global)  
Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this function. At present, $scale$ is always returned as $1.0$, it is returned here to allow for future enhancement.

$info$  
(global)  
If $info = 0$, the execution is successful. If $info < 0$, if the $i$-th argument is an array and the $j$-th entry, indexed $j - 1$, had an illegal value, then $info = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?hegst**

Reduces a Hermitian positive-definite generalized eigenvalue problem to the standard form.

**Syntax**

```c
void pchegst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *descb , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *desca , float *scale , MKL_INT *info );
void pzhegst (MKL_INT *ibtype , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *descb , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *desca , double *scale , MKL_INT *info );
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `p?hegst` function reduces complex Hermitian positive-definite generalized eigenproblems to the standard form.

In the following sub$(A)$ denotes $A(ia:ia+n-1, ja:ja+n-1)$ and sub$(B)$ denotes $B(ib:ib+n-1, jb:jb+n-1)$. If $ibtype = 1$, the problem is

$\text{sub}(A)\text{x} = \lambda \text{sub}(B)\text{x},$

and sub$(A)$ is overwritten by $\text{inv}(U^H)\text{sub}(A)\text{inv}(U)$, or $\text{inv}(L)\text{sub}(A)\text{inv}(L^H)$. 

4 Intel® Math Kernel Library Developer Reference

1426
If \( ibtype = 2 \) or \( 3 \), the problem is

\[
\text{sub}(A)\cdot\text{sub}(B)\cdot x = \lambda x, \quad \text{or} \quad \text{sub}(B)\cdot\text{sub}(A)\cdot x = \lambda x,
\]
and \( \text{sub}(A) \) is overwritten by \( U^*\text{sub}(A)U^H \), or \( L^H*\text{sub}(A)*L \).

\( \text{sub}(B) \) must have been previously factorized as \( U^H*U \) or \( L*L^H \) by \text{p?potrf}.

**Input Parameters**

- \( ibtype \) (global) Must be 1 or 2 or 3.
  - If \( itype = 1 \), compute \( \text{inv}(U^H)\cdot\text{sub}(A)\cdot\text{inv}(U) \), or \( \text{inv}(L)\cdot\text{sub}(A)\cdot\text{inv}(L^H) \);
  - If \( itype = 2 \) or \( 3 \), compute \( U\cdot\text{sub}(A)\cdot U^H \), or \( L^H*\text{sub}(A)*L \).

- \( uplo \) (global) Must be 'U' or 'L'.
  - If \( uplo = 'U' \), the upper triangle of \( \text{sub}(A) \) is stored and \( \text{sub}(B) \) is factored as \( U^H*U \).
  - If \( uplo = 'L' \), the lower triangle of \( \text{sub}(A) \) is stored and \( \text{sub}(B) \) is factored as \( L*L^H \).

- \( n \) (global) The order of the matrices \( \text{sub}(A) \) and \( \text{sub}(B) \) (\( n \geq 0 \)).

- \( a \) (local)

  Pointer into the local memory to an array of size \( lld_a*\text{LOCc}(ja+n-1) \). On entry, the array contains the local pieces of the \( n \)-by-\( n \) Hermitian distributed matrix \( \text{sub}(A) \). If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced. If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( \text{sub}(A) \) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

- \( desca \) (global and local) array of size \( dlen_\) . The array descriptor for the distributed matrix \( A \).

- \( b \) (local)

  Pointer into the local memory to an array of size \( lld_b*\text{LOCc}(jb+n-1) \). On entry, the array contains the local pieces of the triangular factor from the Cholesky factorization of \( \text{sub}(B) \) as returned by \text{p?potrf}.

- \( ib, jb \) (global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the submatrix \( B \), respectively.

- \( descb \) (global and local) array of size \( dlen_\) . The array descriptor for the distributed matrix \( B \).

**Output Parameters**

- \( a \)
  - On exit, if \( info = 0 \), the transformed matrix, stored in the same format as \( \text{sub}(A) \).

- \( scale \) (global)
Amount by which the eigenvalues should be scaled to compensate for the scaling performed in this function. At present, `scale` is always returned as 1.0, it is returned here to allow for future enhancement.

`info` (global)
If `info = 0`, the execution is successful. If `info < 0`, if the i-th argument is an array and the j-th entry, indexed `j - 1`, had an illegal value, then `info = -(i*100+j)`; if the i-th argument is a scalar and had an illegal value, then `info = -i`.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Driver Routines

Table "ScaLAPACK Driver Routines" lists ScaLAPACK driver routines available for solving systems of linear equations, linear least-squares problems, standard eigenvalue and singular value problems, and generalized symmetric definite eigenproblems.

<table>
<thead>
<tr>
<th>Type of Problem</th>
<th>Matrix type, storage scheme</th>
<th>Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear equations</td>
<td>general (partial pivoting)</td>
<td><code>p?gesv</code> (simple driver) / <code>p?gesvx</code> (expert driver)</td>
</tr>
<tr>
<td></td>
<td>general band (partial pivoting)</td>
<td><code>p?qbsv</code> (simple driver)</td>
</tr>
<tr>
<td></td>
<td>general band (no pivoting)</td>
<td><code>p?dbsv</code> (simple driver)</td>
</tr>
<tr>
<td></td>
<td>general tridiagonal (no pivoting)</td>
<td><code>p?dtsv</code> (simple driver)</td>
</tr>
<tr>
<td>Symmetric eigenvalue problem</td>
<td>symmetric/Hermitian positive-definite,</td>
<td><code>p?posv</code> (simple driver) / <code>p?posvx</code> (expert driver)</td>
</tr>
<tr>
<td>Symmetric eigenvalue problem</td>
<td>symmetric/Hermitian, band</td>
<td><code>p?psv</code> (simple driver)</td>
</tr>
<tr>
<td>Symmetric eigenvalue problem</td>
<td>symmetric/Hermitian, tridiagonal</td>
<td><code>p?ptsv</code> (simple driver)</td>
</tr>
<tr>
<td>Generalized symmetric definite eigenvalue problem</td>
<td>general m-by-n</td>
<td><code>p?qgels</code></td>
</tr>
<tr>
<td>Generalized symmetric definite eigenvalue problem</td>
<td>symmetric/Hermitian</td>
<td><code>p?syev / p?heev</code> (simple driver); <code>p?syevd / p?heevd</code> (simple driver with a divide and conquer algorithm); <code>p?syevx / p?heevx</code> (expert driver); <code>p?syevr / p?heevr</code> (simple driver with MRRR algorithm)</td>
</tr>
<tr>
<td>Generalized symmetric definite eigenvalue problem</td>
<td>symmetric/Hermitian</td>
<td><code>p?qgesvd</code></td>
</tr>
<tr>
<td>Generalized symmetric definite eigenvalue problem</td>
<td>general m-by-n</td>
<td><code>p?sygvx / p?hgevx</code> (expert driver)</td>
</tr>
</tbody>
</table>

`p?gesv`

Computes the solution to the system of linear equations with a square distributed matrix and multiple right-hand sides.

Syntax

```c
void psgesv (MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desc , MKL_INT *ipiv , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
```
void pdgesv (MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);

void pcgesv (MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);

void pzgesv (MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *info);

**Include Files**

- mkl_scalapack.h

**Description**

The `p?gesv` function computes the solution to a real or complex system of linear equations \( \text{sub}(A) \cdot X = \text{sub}(B) \), where \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \) is an \( n \)-by-\( n \) distributed matrix and \( X \) and \( \text{sub}(B) = B(ib:ib+n-1, jb:jb+nrhs-1) \) are \( n \)-by-\( nrhs \) distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor \( \text{sub}(A) \) as \( \text{sub}(A) = P \cdot L \cdot U \), where \( P \) is a permutation matrix, \( L \) is unit lower triangular, and \( U \) is upper triangular. \( L \) and \( U \) are stored in \( \text{sub}(A) \). The factored form of \( \text{sub}(A) \) is then used to solve the system of equations \( \text{sub}(A) \cdot X = \text{sub}(B) \).

**Input Parameters**

- \( n \) (global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix \( \text{sub}(A) \) \( (n \geq 0) \).
- \( nrhs \) (global) The number of right hand sides, that is, the number of columns of the distributed submatrices \( B \) and \( X \) \( (nrhs \geq 0) \).
- \( a, b \) (local) Pointers into the local memory to arrays of local size \( a: lld_a \cdot \text{LOCc}(ja+n-1) \) and \( b: lld_b \cdot \text{LOCc}(jb+nrhs-1) \), respectively.
- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of \( \text{sub}(A) \), respectively.
- \( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).
- \( ib, jb \) (global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of \( \text{sub}(B) \), respectively.
- \( descb \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( B \).
Output Parameters

- $a$ Overwritten by the factors $L$ and $U$ from the factorization $\text{sub}(A) = P^T L U$; the unit diagonal elements of $L$ are not stored.
- $b$ Overwritten by the solution distributed matrix $X$.
- $\text{ipiv}$ (local) Array of size $\text{LOCr}(m_a)+mb_a$. This array contains the pivoting information. The (local) row $i$ of the matrix was interchanged with the (global) row $\text{ipiv}[i-1]$.
- $\text{info}$ (global) If $\text{info}=0$, the execution is successful.
  - $\text{info} < 0$: If the $i$-th argument is an array and the $j$-th entry had an illegal value, then $\text{info} = -(i*100+j)$; if the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.
  - $\text{info} > 0$: If $\text{info} = k$, $U(ia+k-1,ja+k-1)$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution could not be computed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?gesvx**

Uses the LU factorization to compute the solution to the system of linear equations with a square matrix $A$ and multiple right-hand sides, and provides error bounds on the solution.

**Syntax**

```c
void pgesvx (char *fact, char *trans, MKL_INT *n, MKL_INT *nrhs, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *af, MKL_INT *iaf, MKL_INT *jaf, MKL_INT *descf, MKL_INT *ipiv, char *equed, float *r, float *c, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *rcond, float *ferr, float *berr, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);
void pdgesvx (char *fact, char *trans, MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *af, MKL_INT *iaf, MKL_INT *jaf, MKL_INT *descf, MKL_INT *ipiv, char *equed, double *r, double *c, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *rcond, double *ferr, double *berr, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);
void pcgesvx (char *fact, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *af, MKL_INT *iaf, MKL_INT *jaf, MKL_INT *descf, MKL_INT *ipiv, char *equed, float *r, float *c, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *rcond, float *ferr, float *berr, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *lrwork, MKL_INT *info);
```
void pzgesvx (char *fact, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *af, MKL_INT *iaf, MKL_INT *jaf, MKL_INT *descaf, MKL_INT *ipiv, char *equed, double *r, double *c, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *rcond, double *ferr, double *berr, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *lrwork, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The pzgesvx function uses the LU factorization to compute the solution to a real or complex system of linear equations $AX = B$, where $A$ denotes the $n$-by-$n$ submatrix $A(ia:ia+n-1, ja:ja+n-1)$, $B$ denotes the $n$-by-$nrhs$ submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$ and $X$ denotes the $n$-by-$nrhs$ submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$.

Error bounds on the solution and a condition estimate are also provided.

In the following description, $af$ stands for the subarray of $af$ from row $iaf$ and column $jaf$ to row $iaf+n-1$ and column $jaf+n-1$.

The function pzgesvx performs the following steps:

1. If fact = 'E', real scaling factors $R$ and $C$ are computed to equilibrate the system:
   
   - $trans = 'N'$: $\text{diag}(R) * A * \text{diag}(C) * (\text{diag}(C))^{-1} * X = \text{diag}(R) * B$
   - $trans = 'T'$: $(\text{diag}(R) * A * \text{diag}(C))^{T} * (\text{diag}(R))^{-1} * X = \text{diag}(C) * B$
   - $trans = 'C'$: $(\text{diag}(R) * A * \text{diag}(C))^{H} * (\text{diag}(R))^{-1} * X = \text{diag}(C) * B$

   Whether or not the system will be equilibrated depends on the scaling of the matrix $A$, but if equilibration is used, $A$ is overwritten by $\text{diag}(R) * A * \text{diag}(C)$ and $B$ by $\text{diag}(R) * B$ (if $trans = 'N'$) or $\text{diag}(C) * B$ (if $trans = 'T'$ or 'C').

2. If fact = 'N' or 'E', the LU decomposition is used to factor the matrix $A$ (after equilibration if fact = 'E') as $A = PLU$, where $P$ is a permutation matrix, $L$ is a unit lower triangular matrix, and $U$ is upper triangular.

3. The factored form of $A$ is used to estimate the condition number of the matrix $A$. If the reciprocal of the condition number is less than relative machine precision, steps 4 - 6 are skipped.

4. The system of equations is solved for $X$ using the factored form of $A$.

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix $X$ is premultiplied by $\text{diag}(C)$ (if $trans = 'N'$) or $\text{diag}(R)$ (if $trans = 'T'$ or 'C') so that it solves the original system before equilibration.

Input Parameters

- $fact$ (global) Must be 'F', 'N', or 'E'.

   Specifies whether or not the factored form of the matrix $A$ is supplied on entry, and if not, whether the matrix $A$ should be equilibrated before it is factored.

   - If $fact = 'F'$ then, on entry, $af$ and $ipiv$ contain the factored form of $A$. If equed is not 'N', the matrix $A$ has been equilibrated with scaling factors given by $r$ and $c$. Arrays $a$, $af$, and $ipiv$ are not modified.
   - If $fact = 'N'$, the matrix $A$ is copied to $af$ and factored.
If $\text{fact} = \text{'E'}$, the matrix $A$ is equilibrated if necessary, then copied to $af$ and factored.

**trans**

(global) Must be 'N', 'T', or 'C'.

Specifies the form of the system of equations:

If trans = 'N', the system has the form $A\times X = B$ (No transpose);

If trans = 'T', the system has the form $A^T\times X = B$ (Transpose);

If trans = 'C', the system has the form $A^H\times X = B$ (Conjugate transpose);

**n**

(global) The number of linear equations; the order of the submatrix $A$ ($n \geq 0$).

**nrhs**

(global) The number of right hand sides; the number of columns of the distributed submatrices $B$ and $X$ ($nrhs \geq 0$).

**a, af, b, work**

(local)

Pointers into the local memory to arrays of local size $a$: $\text{ld}_a \times \text{LOCc}(ja + n - 1)$, $af$: $\text{ld}_af \times \text{LOCc}(ja + n - 1)$, $b$: $\text{ld}_b \times \text{LOCc}(jb + nrhs - 1)$, work: $lwork$.

The array $a$ contains the matrix $A$. If $\text{fact} = \text{'F'}$ and $\text{equed}$ is not 'N', then $A$ must have been equilibrated by the scaling factors in $r$ and/or $c$.

The array $af$ is an input argument if $\text{fact} = \text{'F'}$. In this case it contains on entry the factored form of the matrix $A$, that is, the factors $L$ and $U$ from the factorization $A = P \times L \times U$ as computed by $\text{p?getrf}$. If $\text{equed}$ is not 'N', then $af$ is the factored form of the equilibrated matrix $A$.

The array $b$ contains on entry the matrix $B$ whose columns are the right-hand sides for the systems of equations.

$work$ is a workspace array. The size of $work$ is ($lwork$).

**ia, ja**

(global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A(ia:ia+n-1, ja:ja+n-1)$, respectively.

**desca**

(global and local) array of size $\text{dlen}_a$. The array descriptor for the distributed matrix $A$.

**iaf, jaf**

(global) The row and column indices in the global matrix $AF$ indicating the first row and the first column of the subarray $af$, respectively.

**descaf**

(global and local) array of size $\text{dlen}_a$. The array descriptor for the distributed matrix $AF$.

**ib, jb**

(global) The row and column indices in the global matrix $B$ indicating the first row and the first column of the submatrix $B(ib:ib+n-1, jb:jb+nrhs-1)$, respectively.

**descb**

(global and local) array of size $\text{dlen}_b$. The array descriptor for the distributed matrix $B$.

**ipiv**

(local) Array of size $\text{LOCr}(m_a) + mb_a$.

The array $ipiv$ is an input argument if $\text{fact} = \text{'F'}$.
On entry, it contains the pivot indices from the factorization $A = P^*L^*U$ as computed by `p?getrf`; (local) row $i$ of the matrix was interchanged with the (global) row $ipiv[i - 1]$.

This array must be aligned with $A(is:is+n-1, \ast)$.

**equed**

(global) Must be 'N', 'R', 'C', or 'B'. `equed` is an input argument if `fact` = 'F'. It specifies the form of equilibration that was done:

If `equed` = 'N', no equilibration was done (always true if `fact` = 'N');

If `equed` = 'R', row equilibration was done, that is, $A$ has been premultiplied by $\text{diag}(r)$;

If `equed` = 'C', column equilibration was done, that is, $A$ has been postmultiplied by $\text{diag}(c)$;

If `equed` = 'B', both row and column equilibration was done; $A$ has been replaced by $\text{diag}(r) * A * \text{diag}(c)$.

**r, c**

Arrays of size $\text{LOCr}(n_a)$ and $\text{LOCc}(n_a)$, respectively.

The array $r$ contains the row scale factors for $A$, and the array $c$ contains the column scale factors for $A$. These arrays are input arguments if `fact` = 'F' only; otherwise they are output arguments. If `equed` = 'R' or 'B', $A$ is multiplied on the left by $\text{diag}(r)$; if `equed` = 'N' or 'C', $r$ is not accessed.

If `fact` = 'F' and `equed` = 'R' or 'B', each element of $r$ must be positive.

If `equed` = 'C' or 'B', $A$ is multiplied on the right by $\text{diag}(c)$; if `equed` = 'N' or 'R', $c$ is not accessed.

If `fact` = 'F' and `equed` = 'C' or 'B', each element of $c$ must be positive. Array $r$ is replicated in every process column, and is aligned with the distributed matrix $A$. Array $c$ is replicated in every process row, and is aligned with the distributed matrix $A$.

**ix, jx**

(global) The row and column indices in the global matrix $X$ indicating the first row and the first column of the submatrix $X(ix:ix+n-1, jx:jx+nrhs-1)$, respectively.

**descx**

(global and local) array of size $\text{dlen}_x$. The array descriptor for the distributed matrix $X$.

**lwork**

(local or global) The size of the array `work`; must be at least $\max(p?gecon(lwork), p?gerfs(lwork)) + \text{LOCr}(n_a)$.

**iwork**

(local, pgesvx/pdgesvx only). Workspace array. The size of `iwork` is $(liwork)$.

**liwork**

(local, pgesvx/pdgesvx only). The size of the array `iwork`, must be at least $\text{LOCr}(n_a)$.

**rwork**

(local) Workspace array, used in complex flavors only.
The size of rwork is (lrwork).

\( lrwork \) (local or global, pcgesvx/pzgesvx only). The size of the array rwork; must be at least \( 2 \times \text{LOCc}(n_a) \).

**Output Parameters**

\( x \) (local)

Pointer into the local memory to an array of local size \( \text{lld}_x \times \text{LOCc}(jx +nrhs-1) \).

If info = 0, the array x contains the solution matrix \( X \) to the original system of equations. Note that \( A \) and \( B \) are modified on exit if \( \text{equed} \neq 'N' \), and the solution to the equilibrated system is:

\[
\begin{align*}
\text{diag}(C)-1 & \times X, \text{ if } \text{trans} = 'N' \text{ and } \text{equed} = 'C' \text{ or } 'B'; \text{ and} \\
\text{diag}(R)-1 & \times X, \text{ if } \text{trans} = 'T' \text{ or } 'C' \text{ and } \text{equed} = 'R' \text{ or } 'B'. \\
\end{align*}
\]

Array \( a \) is not modified on exit if \( \text{fact} = 'F' \) or 'N', or if \( \text{fact} = 'E' \) and \( \text{equed} = 'N' \).

If \( \text{equed} \neq 'N' \), \( A \) is scaled on exit as follows:

- \( \text{equed} = 'R': A = \text{diag}(R) \times A \)
- \( \text{equed} = 'C': A = A \times \text{diag}(c) \)
- \( \text{equed} = 'B': A = \text{diag}(R) \times A \times \text{diag}(c) \)

\( af \)

If \( \text{fact} = 'N' \) or 'E', then \( af \) is an output argument and on exit returns the factors \( L \) and \( U \) from the factorization \( A = P \times L \times U \) of the original matrix \( A \) (if \( \text{fact} = 'N' \)) or of the equilibrated matrix \( A \) (if \( \text{fact} = 'E' \)). See the description of \( a \) for the form of the equilibrated matrix.

\( b \)

Overwritten by \( \text{diag}(R) \times B \) if \( \text{trans} = 'N' \) and \( \text{equed} = 'R' \) or 'B'; overwritten by \( \text{diag}(c) \times B \) if \( \text{trans} = 'T' \) and \( \text{equed} = 'C' \) or 'B'; not changed if \( \text{equed} = 'N' \).

\( r, c \)

These arrays are output arguments if \( \text{fact} \neq 'F' \).

See the description of \( r, c \) in Input Arguments section.

\( rcond \)

(global).

An estimate of the reciprocal condition number of the matrix \( A \) after equilibration (if done). The function sets \( rcond = 0 \) if the estimate underflows; in this case the matrix is singular (to working precision). However, anytime \( rcond \) is small compared to 1.0, for the working precision, the matrix may be poorly conditioned or even singular.

\( ferr, berr \)

(local)

Arrays of size \( \text{LOCc}(n_b) \) each. Contain the component-wise forward and relative backward errors, respectively, for each solution vector.

Arrays \( ferr \) and \( berr \) are both replicated in every process row, and are aligned with the matrices \( B \) and \( X \).
If fact = 'N' or 'E', then ipiv is an output argument and on exit contains the pivot indices from the factorization \( A = P*L*U \) of the original matrix \( A \) (if fact = 'N') or of the equilibrated matrix \( A \) (if fact = 'E').

If fact\#'E', then equed is an output argument. It specifies the form of equilibration that was done (see the description of equed in Input Arguments section).

If info=0, on exit work[0] returns the minimum value of lwork required for optimum performance.

If info=0, on exit iwork[0] returns the minimum value of liwork required for optimum performance.

If info=0, on exit rwork[0] returns the minimum value of lrwork required for optimum performance.

If info=0, the execution is successful.

See Also Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?gbsv}

\textit{Computes the solution to the system of linear equations with a general banded distributed matrix and multiple right-hand sides.}

\textbf{Syntax}

```c
void psgbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, float *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, float *b, MKL_INT *ib, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, double *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, double *b, MKL_INT *ib, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```
Include Files
• mkl_scalapack.h

Description
The `p?gbsv` function computes the solution to a real or complex system of linear equations

\[ \text{sub}(A) \times X = \text{sub}(B), \]

where \( \text{sub}(A) = A(1:n, ja:ja+n-1) \) is an \( n \)-by-\( n \) real/complex general banded distributed matrix with \( \text{bwl} \) subdiagonals and \( \text{bwu} \) superdiagonals, and \( X \) and \( \text{sub}(B) = B(ib:ib+n-1, 1:nrhs) \) are \( n \)-by-\( nrhs \) distributed matrices.

The LU decomposition with partial pivoting and row interchanges is used to factor \( \text{sub}(A) \) as \( \text{sub}(A) = P \times L \times U \times Q \), where \( P \) and \( Q \) are permutation matrices, and \( L \) and \( U \) are banded lower and upper triangular matrices, respectively. The matrix \( Q \) represents reordering of columns for the sake of parallelism, while \( P \) represents reordering of rows for numerical stability using classic partial pivoting.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\( n \)  
(global) The number of rows and columns to be operated on, that is, the order of the distributed matrix \( \text{sub}(A) \) (\( n \geq 0 \)).

\( \text{bwl} \)  
(global) The number of subdiagonals within the band of \( A \) (\( 0 \leq \text{bwl} \leq n-1 \)).

\( \text{bwu} \)  
(global) The number of superdiagonals within the band of \( A \) (\( 0 \leq \text{bwu} \leq n-1 \)).

\( nrhs \)  
(global) The number of right hand sides; the number of columns of the distributed matrix \( \text{sub}(B) \) (\( nrhs \geq 0 \)).

\( a, b \)  
(local) Pointers into the local memory to arrays of local size \( a: lld_a*\text{LOCc}(ja+n-1) \) and \( b: lld_b*\text{LOCc}(nrhs) \).

On entry, the array \( a \) contains the local pieces of the global array \( A \).

On entry, the array \( b \) contains the right hand side distributed matrix \( \text{sub}(B) \).

\( ja \)  
(global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( \text{desca} \)  
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

If \( \text{desca}[\text{dtype}_-1] = 501 \), then \( dlen \geq 7 \);
else if \( \text{desca}[\text{dtype} - 1] = 1 \), then \( dlen \geq 9 \).

\( \text{ib} \)

(global) The row index in the global matrix \( B \) indicating the first row of the matrix to be operated on (which may be either all of \( B \) or a submatrix of \( B \)).

\( \text{descb} \)

(global and local) array of size \( dlen \). The array descriptor for the distributed matrix \( B \).

If \( \text{descb}[\text{dtype} - 1] = 502 \), then \( dlen \geq 7 \);

else if \( \text{descb}[\text{dtype} - 1] = 1 \), then \( dlen \geq 9 \).

\( \text{work} \)

(local)

Workspace array of size \( lwork \).

\( lwork \)

(local or global) The size of the array \( \text{work} \), must be at least \( lwork \geq (NB + bwu) \times (bwl + bwu) + 6 \times (bwl + bwu) \times (bwl + 2 \times bwu) + \max(nrhs \times (NB + 2 \times bwl + 4 \times bwu), 1) \).

Output Parameters

\( a \)

On exit, contains details of the factorization. Note that the resulting factorization is not the same factorization as returned from LAPACK. Additional permutations are performed on the matrix for the sake of parallelism.

\( b \)

On exit, this array contains the local pieces of the solution distributed matrix \( X \).

\( \text{ipiv} \)

(local) array.

The size of \( \text{ipiv} \) must be at least \( \text{desca}[\text{NB} - 1] \). This array contains pivot indices for local factorizations. You should not alter the contents between factorization and solve.

\( \text{work[0]} \)

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( \text{info} \)

If \( \text{info}=0 \), the execution is successful. \( \text{info} < 0 \):

If the \( i \)-th argument is an array and the \( j \)-th entry had an illegal value, then \( \text{info} = -(i \times 100 + j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

\( \text{info} > 0 \):

If \( \text{info} = k \leq \text{NPROCS} \), the submatrix stored on processor \( \text{info} \) and factored locally was not nonsingular, and the factorization was not completed. If \( \text{info} = k > \text{NPROCS} \), the submatrix stored on processor \( \text{info} - \text{NPROCS} \) representing interactions with other processors was not nonsingular, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\( \text{p?dbsv} \)

Solves a general band system of linear equations.
Syntax

void psdbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, float *a, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info );

void pddbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, double *a, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info );

void pcdbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info );

void pzdbsv (MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info );

Include Files

• mkl_scalapack.h

Description

The p?dbsv function solves the following system of linear equations:

\[ A(1:n, ja:ja+n-1) \times X = B(ib:ib+n-1, 1:nrhs), \]

where \( A(1:n, ja:ja+n-1) \) is an \( n \)-by-\( n \) real/complex banded diagonally dominant-like distributed matrix with bandwidth \( bwl, bwu \).

Gaussian elimination without pivoting is used to factor a reordering of the matrix into \( LU \).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\( n \) (global) The order of the distributed submatrix \( A \), \( n \geq 0 \).

\( bwl \) (global) Number of subdiagonals. \( 0 \leq bwl \leq n-1 \).

\( bwu \) (global) Number of subdiagonals. \( 0 \leq bwu \leq n-1 \).

\( nrhs \) (global) The number of right-hand sides; the number of columns of the distributed submatrix \( B \), \( nrhs \geq 0 \).

\( a \) (local).
Pointer into the local memory to an array with leading size \( lld\_a \geq (bwl + bwu+1) \) (stored in \( desca \)). On entry, this array contains the local pieces of the distributed matrix.

\( ja \)

(global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( desca \)

(global and local) array of size \( dlen \).
If 1d type (\( dtype\_a = 501 \) or 502), \( dlen \geq 7 \);
If 2d type (\( dtype\_a = 1 \)), \( dlen \geq 9 \).
The array descriptor for the distributed matrix \( A \).
Contains information of mapping of \( A \) to memory.

\( b \)

(local)
Pointer into the local memory to an array of local lead size \( lld\_b \geq nb \). On entry, this array contains the local pieces of the right hand sides \( B(ib:ib + n-1, 1:nrhs) \).

\( ib \)

(global) The row index in the global matrix \( B \) indicating the first row of the matrix to be operated on (which may be either all of \( b \) or a submatrix of \( B \)).

\( descb \)

(global and local) array of size \( dlen \).
If 1d type (\( dtype\_b = 502 \)), \( dlen \geq 7 \);
If 2d type (\( dtype\_b = 1 \)), \( dlen \geq 9 \).
The array descriptor for the distributed matrix \( B \).
Contains information of mapping of \( B \) to memory.

\( work \)

(local).
Temporary workspace. This space may be overwritten in between calls to functions. \( work \) must be the size given in \( lwork \).

\( lwork \)

(local or global) Size of user-input workspace \( work \). If \( lwork \) is too small, the minimal acceptable size will be returned in \( work[0] \) and an error code is returned.

\[ lwork \geq nb(bwi+bwu)+6\max(bwl,bwu)\max(bwl,bwu)+\max((\max(bwl,bwu)\max\( bwl,bwu)), \max(bwl,bwu)\max(bwl,bwu)) \]

**Output Parameters**

\( a \)

On exit, this array contains information containing details of the factorization.
Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

\( b \)

On exit, this contains the local piece of the solutions distributed matrix \( X \).

\( work \)

On exit, \( work[0] \) contains the minimal \( lwork \).

\( info \)

(local) If \( info=0 \), the execution is successful.
If the \(i\)-th argument is an array and the \(j\)-entry had an illegal value, then \(info = -(i*100+j)\), if the \(i\)-th argument is a scalar and had an illegal value, then \(info = -i\).

> 0: If \(info = k < \text{NPROCS}\), the submatrix stored on processor \(info\) and factored locally was not positive definite, and the factorization was not completed.

If \(info = k > \text{NPROCS}\), the submatrix stored on processor \(info - \text{NPROCS}\) representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?dtsv}

Solves a general tridiagonal system of linear equations.

\textbf{Syntax}

\begin{verbatim}
void psdtsv (MKL_INT *n, MKL_INT *nrhs, float *dl, float *d, float *du, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pddtsv (MKL_INT *n, MKL_INT *nrhs, double *dl, double *d, double *du, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
void pcdtsv (MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzdtsv (MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
\end{verbatim}

\textbf{Include Files}

- mkl_scalapack.h

\textbf{Description}

The function solves a system of linear equations

\[ A(1:n, ja:ja+n-1) * X = B(ib:ib+n-1, 1:nrhs), \]

where \(A(1:n, ja:ja+n-1)\) is an \(n\)-by-\(n\) complex tridiagonal diagonally dominant-like distributed matrix. Gaussian elimination without pivoting is used to factor a reordering of the matrix into \(L U\).
Optimization Notice

Optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\( n \) (global) The order of the distributed submatrix \( A(n \geq 0) \).

\( nrhs \)

The number of right hand sides; the number of columns of the distributed matrix \( B(nrhs \geq 0) \).

\( dl \) (local).

Pointer to local part of global vector storing the lower diagonal of the matrix. Globally, \( dl[0] \) is not referenced, and \( dl \) must be aligned with \( d \).

Must be of size \( > desca[\text{nb}_- - 1] \).

\( d \) (local).

Pointer to local part of global vector storing the main diagonal of the matrix.

\( du \) (local).

Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, \( du[n - 1] \) is not referenced, and \( du \) must be aligned with \( d \).

\( ja \) (global) The index in the global matrix \( A \) indicating the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( desca \) (global and local) array of size \( dlen \).

If 1d type (\( \text{dtype}_a=501 \) or 502), \( dlen \geq 7 \);  
If 2d type (\( \text{dtype}_a=1 \)), \( dlen \geq 9 \).

The array descriptor for the distributed matrix \( A \).

Contains information of mapping of \( A \) to memory.

\( b \) (local)

Pointer into the local memory to an array of local lead size \( lld_b > \text{nb} \). On entry, this array contains the local pieces of the right hand sides \( B(\text{ib}:\text{ib} + n-1, 1:nh\text{rhs}) \).

\( ib \) (global) The row index in the global matrix \( B \) indicating the first row of the matrix to be operated on (which may be either all of \( B \) or a submatrix of \( B \)).

\( descb \) (global and local) array of size \( dlen \).

If 1d type (\( \text{dtype}_b =502 \)), \( dlen \geq 7 \);  
If 2d type (\( \text{dtype}_b =1 \)), \( dlen \geq 9 \).

The array descriptor for the distributed matrix \( B \).

Contains information of mapping of \( B \) to memory.

\( work \) (local).
lwork

(local or global) Size of user-input workspace work. If lwork is too small, the minimal acceptable size will be returned in work[0] and an error code is returned. lwork > (12*NPCOL+3*nb)+max((10+2*min(100, nrhs))*NP COL+4*nrhs, 8*NP COL)

Output Parameters

dl

On exit, this array contains information containing the * factors of the matrix.

d

On exit, this array contains information containing the * factors of the matrix. Must be of size > desca[nb_ - 1].

du

On exit, this array contains information containing the * factors of the matrix. Must be of size > desca[nb_ - 1].

b

On exit, this contains the local piece of the solutions distributed matrix X.

work

On exit, work[0] contains the minimal lwork.

info

(output) If info=0, the execution is successful.

< 0: If the i-th argument is an array and the j-entry had an illegal value, then info = -(i*100+j), if the j-th argument is a scalar and had an illegal value, then info = -i.

> 0: If info = k<NPROCS, the submatrix stored on processor info and factored locally was not positive definite, and the factorization was not completed.

If info = k > NPROCS, the submatrix stored on processor info-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?posv

Solves a symmetric positive definite system of linear equations.

Syntax

void psposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pdposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pcposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );

void pzposv (char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *info );
Include Files
• mkl_scalapack.h

Description
The p?posv function computes the solution to a real/complex system of linear equations
sub(A) * X = sub(B),
where sub(A) denotes \( A(ia:ia+n-1, ja:ja+n-1) \) and is an \( n \)-by-\( n \) symmetric/Hermitian distributed positive definite matrix and \( X \) and sub(\( B \)) denoting \( B(ib:ib+n-1, jb:jb+nrhs-1) \) are \( n \)-by-\( nrhs \) distributed matrices. The Cholesky decomposition is used to factor sub(A) as
\[
\begin{align*}
\text{sub}(A) &= U^T \ast U, & \text{if } \text{uplo} = 'U' , \\
\text{sub}(A) &= L \ast L^T, & \text{if } \text{uplo} = 'L',
\end{align*}
\]
where \( U \) is an upper triangular matrix and \( L \) is a lower triangular matrix. The factored form of sub(A) is then used to solve the system of equations.

Input Parameters

uplo (global) Must be 'U' or 'L'.
Indicates whether the upper or lower triangular part of sub(A) is stored.

n (global) The order of the distributed matrix sub(A) \( (n \geq 0) \).

nrhs
The number of right-hand sides; the number of columns of the distributed matrix sub(\( B \)) \( (nrhs \geq 0) \).

a (local)
Pointer into the local memory to an array of size \( lld_a^* \text{LOCc}(ja+n-1) \). On entry, this array contains the local pieces of the \( n \)-by-\( n \) symmetric distributed matrix sub(A) to be factored.

If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of sub(A) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of sub(A) contains the lower triangular part of the distributed matrix, and its strictly upper triangular part is not referenced.

ia, ja (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

desca (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

b (local)
Pointer into the local memory to an array of size \( lld_b^* \text{LOCc}(jb+nrhs-1) \). On entry, the local pieces of the right hand sides distributed matrix sub(\( B \)).

ib, jb (global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the submatrix \( B \), respectively.

descb (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( B \).
**Output Parameters**

- **a**
  
  On exit, if \( \text{info} = 0 \), this array contains the local pieces of the factor \( U \) or \( L \) from the Cholesky factorization \( \text{sub}(A) = U^T U \) or \( L^H L \).

- **b**
  
  On exit, if \( \text{info} = 0 \), \( \text{sub}(B) \) is overwritten by the solution distributed matrix \( X \).

- **info**

  (global)

  If \( \text{info} = 0 \), the execution is successful.

  If \( \text{info} < 0 \): If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

  If \( \text{info} > 0 \): If \( \text{info} = k \), the leading minor of order \( k \), \( A(ia:ia+k-1, ja:ja+k-1) \) is not positive definite, and the factorization could not be completed, and the solution has not been computed.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?posvx**

Solves a symmetric or Hermitian positive definite system of linear equations.

**Syntax**

```c
void p prima (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , char *equed , float *sr , float *sc , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , float *rcond , float *ferr , float *berr , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *info );
```

```c
void pdposvx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , char *equed , double *sr , double *sc , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double *berr , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *info );
```

```c
void pcpmx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , char *equed , float *sr , float *sc , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex8 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *info );
```

```c
void pcpmx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , char *equed , double *sr , double *sc , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *info );
```

```c
void pcpmx (char *fact , char *uplo , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *af , MKL_INT *iaf , MKL_INT *jaf , MKL_INT *descaf , char *equed , double *sr , double *sc , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *x , MKL_INT *ix , MKL_INT *jx , MKL_INT *descx , double *rcond , double *ferr , double *berr , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *info );
```
Include Files
• mkl_scalapack.h

Description
The p?posvx function uses the Cholesky factorization \( A = U^T U \) or \( A = L L^T \) to compute the solution to a real or complex system of linear equations

\[
A(ia:ia+n-1, ja:ja+n-1)X = B(ib:ib+n-1, jb:jb+nrhs-1),
\]

where \( A(ia:ia+n-1, ja:ja+n-1) \) is a \( n \times n \) matrix and \( X \) and \( B(ib:ib+n-1, jb:jb+nrhs-1) \) are \( n \times nrhs \) matrices.

Error bounds on the solution and a condition estimate are also provided.

In the following comments \( y \) denotes \( Y(iy:iy+m-1, jy:jy+k-1) \), an \( m \times k \) matrix where \( y \) can be \( a, af, b \) and \( x \).

The function p?posvx performs the following steps:

1. If \( \text{fact} = 'E' \), real scaling factors \( s \) are computed to equilibrate the system:

\[
\text{diag}(sr) * A * \text{diag}(sc) * \text{inv(diag(sc)}) * X = \text{diag}(sr) * B
\]

Whether or not the system will be equilibrated depends on the scaling of the matrix \( A \), but if equilibration is used, \( A \) is overwritten by \( \text{diag}(sr) * A * \text{diag}(sc) \) and \( B \) by \( \text{diag}(sr) * B \).

2. If \( \text{fact} = 'N' \) or \( 'E' \), the Cholesky decomposition is used to factor the matrix \( A \) (after equilibration if \( \text{fact} = 'E' \)) as

\[
A = U^T U, \text{ if } \text{uplo} = 'U', \text{ or } A = L L^T, \text{ if } \text{uplo} = 'L',
\]

where \( U \) is an upper triangular matrix and \( L \) is a lower triangular matrix.

3. The factored form of \( A \) is used to estimate the condition number of the matrix \( A \). If the reciprocal of the condition number is less than machine precision, steps 4-6 are skipped.

4. The system of equations is solved for \( X \) using the factored form of \( A \).

5. Iterative refinement is applied to improve the computed solution matrix and calculate error bounds and backward error estimates for it.

6. If equilibration was used, the matrix \( X \) is premultiplied by \( \text{diag}(sr) \) so that it solves the original system before equilibration.

Input Parameters

\( \text{fact} \) (global) Must be \( 'F', 'N', \) or \( 'E' \).

Specifies whether or not the factored form of the matrix \( A \) is supplied on entry, and if not, whether the matrix \( A \) should be equilibrated before it is factored.

If \( \text{fact} = 'F' \): on entry, \( af \) contains the factored form of \( A \). If \( \text{equed} = 'Y' \), the matrix \( A \) has been equilibrated with scaling factors given by \( s. a \) and \( af \) will not be modified.

If \( \text{fact} = 'N' \), the matrix \( A \) will be copied to \( af \) and factored.

If \( \text{fact} = 'E' \), the matrix \( A \) will be equilibrated if necessary, then copied to \( af \) and factored.

\( \text{uplo} \) (global) Must be \( 'U' \) or \( 'L' \).

Indicates whether the upper or lower triangular part of \( A \) is stored.
(global) The order of the distributed matrix sub(A) \( (n \geq 0) \).

\( nrhs \) (global) The number of right-hand sides; the number of columns of the distributed submatrices \( B \) and \( X \). \( (nrhs \geq 0) \).

\( a \) (local) Pointer into the local memory to an array of local size \( lld_a \cdot \text{LOCc}(ja + n - 1) \). On entry, the symmetric/Hermitian matrix \( A \), except if \( \text{fact} = 'F' \) and \( \text{equed} = 'Y' \), then \( A \) must contain the equilibrated matrix \( \text{diag}(sr) \cdot A \cdot \text{diag}(sc) \).

If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( A \) contains the upper triangular part of the matrix \( A \), and the strictly lower triangular part of \( A \) is not referenced.

If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( A \) contains the lower triangular part of the matrix \( A \), and the strictly upper triangular part of \( A \) is not referenced. \( A \) is not modified if \( \text{fact} = 'F' \) or \( 'N' \), or if \( \text{fact} = 'E' \) and \( \text{equed} = 'N' \) on exit.

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

\( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( af \) (local) Pointer into the local memory to an array of local size \( lld_{af} \cdot \text{LOCc}(ja + n - 1) \).

If \( \text{fact} = 'F' \), then \( af \) is an input argument and on entry contains the triangular factor \( U \) or \( L \) from the Cholesky factorization \( A = U^T \cdot U \) or \( A = L \cdot L^T \), in the same storage format as \( A \). If \( \text{equed} \neq 'N' \), then \( af \) is the factored form of the equilibrated matrix \( \text{diag}(sr) \cdot A \cdot \text{diag}(sc) \).

\( iaf, jaf \) (global) The row and column indices in the global matrix \( AF \) indicating the first row and the first column of the submatrix \( AF \), respectively.

\( descaf \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( AF \).

\( equed \) (global) Must be \( 'N' \) or \( 'Y' \).

\( equed \) is an input argument if \( \text{fact} = 'F' \). It specifies the form of equilibration that was done:

- If \( \text{equed} = 'N' \), no equilibration was done (always true if \( \text{fact} = 'N' \));
- If \( \text{equed} = 'Y' \), equilibration was done and \( A \) has been replaced by \( \text{diag}(sr) \cdot A \cdot \text{diag}(sc) \).

\( sr \) (local) Array of size \( lld_a \).

The array \( s \) contains the scale factors for \( A \). This array is an input argument if \( \text{fact} = 'F' \) only; otherwise it is an output argument.

If \( \text{equed} = 'N' \), \( s \) is not accessed.
If \( \text{fact} = 'F' \) and \( \text{equed} = 'Y' \), each element of \( s \) must be positive.

\( b \)  
(local)  
Pointer into the local memory to an array of local size \( lld_b \* \text{LOCc}(jb +nrhs-1) \). On entry, the \( n \)-by-\( nrhs \) right-hand side matrix \( B \).

\( ib, jb \)  
(global) The row and column indices in the global matrix \( B \) indicating the first row and the first column of the submatrix \( B \), respectively.

\( descb \)  
(global and local) Array of size \( dlen_ \). The array descriptor for the distributed matrix \( B \).

\( x \)  
(local)  
Pointer into the local memory to an array of local size \( lld_x \* \text{LOCc}(jx +nrhs-1) \).

\( ix, jx \)  
(global) The row and column indices in the global matrix \( X \) indicating the first row and the first column of the submatrix \( X \), respectively.

\( descx \)  
(global and local) Array of size \( dlen_ \). The array descriptor for the distributed matrix \( X \).

\( work \)  
(local)  
Workspace array of size \( lwork \).

\( lwork \)  
(local or global)  
The size of the array \( work \). \( lwork \) is local input and must be at least \( lwork = \max(p \text{-pocon}(lwork), p \text{-porfs}(lwork)) + \text{LOCr}(n_a) \).

\( lwork = 3 \* \text{desca}[lld - 1] \).

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

\( iwork \)  
(local)  
Workspace array of size \( liwork \).

\( liwork \)  
(local or global)  
The size of the array \( iwork \). \( liwork \) is local input and must be at least \( liwork = \text{desca}[lld - 1] \* \text{liwork} = \text{LOCr}(n_a) \).

If \( liwork = -1 \), then \( liwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\( a \)  
On exit, if \( \text{fact} = 'E' \) and \( \text{equed} = 'Y' \), \( a \) is overwritten by \( \text{diag}(sr) \* a \* \text{diag}(sc) \).

\( af \)  
If \( \text{fact} = 'N' \), then \( af \) is an output argument and on exit returns the triangular factor \( U \) or \( L \) from the Cholesky factorization \( A = U^T \* U \) or \( A = L \* L^T \) of the original matrix \( A \).
If \( \text{fact} = 'E' \), then \( af \) is an output argument and on exit returns the triangular factor \( U \) or \( L \) from the Cholesky factorization \( A = U^T U \) or \( A = L^T L \) of the equilibrated matrix \( A \) (see the description of \( A \) for the form of the equilibrated matrix).

**equed**

If \( \text{fact} \neq 'F' \), then \( \text{equed} \) is an output argument. It specifies the form of equilibration that was done (see the description of \( \text{equed} \) in Input Arguments section).

**sr**

This array is an output argument if \( \text{fact} \neq 'F' \).

See the description of \( sr \) in Input Arguments section.

**sc**

This array is an output argument if \( \text{fact} \neq 'F' \).

See the description of \( sc \) in Input Arguments section.

**b**

On exit, if \( \text{equed} = 'N' \), \( b \) is not modified; if \( \text{trans} = 'N' \) and \( \text{equed} = 'R' \) or \( 'B' \), \( b \) is overwritten by \( \text{diag}(r) \cdot b \); if \( \text{trans} = 'T' \) or \( 'C' \) and \( \text{equed} = 'C' \) or \( 'B' \), \( b \) is overwritten by \( \text{diag}(c) \cdot b \).

**x**

If \( \text{info} = 0 \) the \( n \)-by-\( nrhs \) solution matrix \( X \) to the original system of equations.

Note that \( A \) and \( B \) are modified on exit if \( \text{equed} \neq 'N' \), and the solution to the equilibrated system is

\[
\text{inv(diag(sc))} \cdot X \quad \text{if trans} = 'N' \quad \text{and} \quad \text{equed} = 'C' \quad \text{or} \quad 'B',
\]

or

\[
\text{inv(diag(sr))} \cdot X \quad \text{if trans} = 'T' \quad \text{or} \quad 'C' \quad \text{and} \quad \text{equed} = 'R' \quad \text{or} \quad 'B'.
\]

**rcond**

An estimate of the reciprocal condition number of the matrix \( A \) after equilibration (if done). If \( rcond \) is less than the machine precision (in particular, if \( rcond=0 \)), the matrix is singular to working precision. This condition is indicated by a return code of \( \text{info} > 0 \).

**ferr**

Arrays of size at least \( \max(LOC,n_b) \). The estimated forward error bounds for each solution vector \( X(j) \) (the \( j \)-th column of the solution matrix \( X \)). If \( xtrue \) is the true solution, \( ferr[j - 1] \) bounds the magnitude of the largest entry in \( (X(j) - xtrue) \) divided by the magnitude of the largest entry in \( X(j) \). The quality of the error bound depends on the quality of the estimate of \( \text{norm(inv}(A)) \) computed in the code; if the estimate of \( \text{norm(inv}(A)) \) is accurate, the error bound is guaranteed.

**berr**

Arrays of size at least \( \max(LOC,n_b) \). The componentwise relative backward error of each solution vector \( X(j) \) (the smallest relative change in any entry of \( A \) or \( B \) that makes \( X(j) \) an exact solution).

**work[0]**

(\( local \)) On exit, \( \text{work}[0] \) returns the minimal and optimal \( liwork \).

**info**

(\( global \))

If \( \text{info}=0 \), the execution is successful.

\(< 0 \): if \( \text{info} = -i \), the \( i \)-th argument had an illegal value.
> 0: if info = i, and i is ≤ n: if info = i, the leading minor of order i of a is not positive definite, so the factorization could not be completed, and the solution and error bounds could not be computed.

= n+1: rcond is less than machine precision. The factorization has been completed, but the matrix is singular to working precision, and the solution and error bounds have not been computed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?pbsv
Solves a symmetric/Hermitian positive definite banded system of linear equations.

Syntax
void pspbsv (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_INT *nrhs, float *a, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);

void pdpbsv (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_INT *nrhs, double *a, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);

void pcpbsv (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzpbsv (char *uplo, MKL_INT *n, MKL_INT *bw, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?pbsv function solves a system of linear equations
\[ A(1:n, ja:ja+n-1)*X = B(ib:ib+n-1, 1:nrhs), \]
where \( A(1:n, ja:ja+n-1) \) is an \( n \)-by-\( n \) real/complex banded symmetric positive definite distributed matrix with bandwidth \( bw \).

Cholesky factorization is used to factor a reordering of the matrix into \( L*L' \).

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Input Parameters

**uplo**

(global) Must be 'U' or 'L'.

Indicates whether the upper or lower triangular of \(A\) is stored.

If \(uplo = 'U'\), the upper triangular \(A\) is stored

If \(uplo = 'L'\), the lower triangular of \(A\) is stored.

**n**

(global) The order of the distributed matrix \(A\) (\(n \geq 0\)).

**bw**

(global) The number of subdiagonals in \(L\) or \(U\). \(0 \leq bw \leq n-1\).

**nrhs**

(global) The number of right-hand sides; the number of columns in \(B\) (\(nrhs \geq 0\)).

**a**

(local).

Pointer into the local memory to an array with leading size \(lld_a \geq (bw +1)\) (stored in \(desca\)). On entry, this array contains the local pieces of the distributed matrix \(\text{sub}(A)\) to be factored.

**ja**

(global) The index in the global matrix \(A\) indicating the start of the matrix to be operated on (which may be either all of \(A\) or a submatrix of \(A\)).

**desca**

(global and local) array of size \(dlen\). The array descriptor for the distributed matrix \(A\).

**b**

(local)

Pointer into the local memory to an array of local lead size \(lld_b \geq nb\). On entry, this array contains the local pieces of the right hand sides \(B(ib:ib+n-1, 1:nrhs)\).

**ib**

(global) The row index in the global matrix \(B\) indicating the first row of the matrix to be operated on (which may be either all of \(b\) or a submatrix of \(B\)).

**descb**

(global and local) array of size \(dlen\).

If 1D type \((dtype_b = 502)\), \(dlen \geq 7\);

If 2D type \((dtype_b = 1)\), \(dlen \geq 9\).

The array descriptor for the distributed matrix \(B\).

Contains information of mapping of \(B\) to memory.

**work**

(local).

Temporary workspace. This space may be overwritten in between calls to functions. \(work\) must be the size given in \(lwork\).

**lwork**

(local or global) Size of user-input workspace \(work\). If \(lwork\) is too small, the minimal acceptable size will be returned in \(work[0]\) and an error code is returned. \(lwork \geq (nb+2*bw)*bw + \max((bw*nrhs), bw*bw)\)
Output Parameters

- **a**: On exit, this array contains information containing details of the factorization. Note that permutations are performed on the matrix, so that the factors returned are different from those returned by LAPACK.

- **b**: On exit, contains the local piece of the solutions distributed matrix $X$.

- **work**: On exit, `work[0]` contains the minimal `lwork`.

- **info**: (global) If `info`=0, the execution is successful.
  - If the $i$-th argument is an array and the $j$-entry had an illegal value, then `info = -(i*100+j)`, if the $i$-th argument is a scalar and had an illegal value, then `info = -i`.
  - If `info = k ≤ NPROCS`, the submatrix stored on processor `info` and factored locally was not positive definite, and the factorization was not completed.
  - If `info = k > NPROCS`, the submatrix stored on processor `info-NPROCS` representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?ptsv**

**Syntax**

Solves a symmetric or Hermitian positive definite tridiagonal system of linear equations.

```c
void psptsv (MKL_INT *n, MKL_INT *nrhs, float *d, float *e, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pdptsv (MKL_INT *n, MKL_INT *nrhs, double *d, double *e, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
void pcptsv (char *uplo, MKL_INT *n, MKL_INT *nrhs, float *d, MKL_Complex8 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzptsv (char *uplo, MKL_INT *n, MKL_INT *nrhs, double *d, MKL_Complex16 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `p?ptsv` function solves a system of linear equations

$$A(1:n, ja:ja+n-1)X = B(ib:ib+n-1, 1:nrhs),$$

where $A(1:n, ja:ja+n-1)$ is an $n$-by-$n$ real tridiagonal symmetric positive definite distributed matrix.
Cholesky factorization is used to factor a reordering of the matrix into $L^*L'$.

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>(global) The order of matrix $A$ ($n \geq 0$).</td>
</tr>
<tr>
<td>$nrhs$</td>
<td>(global) The number of right-hand sides; the number of columns of the distributed submatrix $B$ ($nrhs \geq 0$).</td>
</tr>
<tr>
<td>$d$</td>
<td>(local) Pointer to local part of global vector storing the main diagonal of the matrix.</td>
</tr>
<tr>
<td>$e$</td>
<td>(local) Pointer to local part of global vector storing the upper diagonal of the matrix. Globally, $du(n)$ is not referenced, and $du$ must be aligned with $d$.</td>
</tr>
<tr>
<td>$ja$</td>
<td>(global) The index in the global matrix $A$ indicating the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).</td>
</tr>
<tr>
<td>$desca$</td>
<td>(global and local) array of size $dlen$. If $1d$ type ($dtype_a=501$ or $502$), $dlen \geq 7$; If $2d$ type ($dtype_a=1$), $dlen \geq 9$. The array descriptor for the distributed matrix $A$. Contains information of mapping of $A$ to memory.</td>
</tr>
<tr>
<td>$b$</td>
<td>(local) Pointer into the local memory to an array of local lead size $lld_b \geq nb$. On entry, this array contains the local pieces of the right hand sides $B(ib:ib+n-1, 1:nrhs)$.</td>
</tr>
<tr>
<td>$ib$</td>
<td>(global) The row index in the global matrix $B$ indicating the first row of the matrix to be operated on (which may be either all of $b$ or a submatrix of $B$).</td>
</tr>
<tr>
<td>$descb$</td>
<td>(global and local) array of size $dlen$. If $1d$ type ($dtype_b = 502$), $dlen \geq 7$; If $2d$ type ($dtype_b = 1$), $dlen \geq 9$. The array descriptor for the distributed matrix $B$. Contains information of mapping of $B$ to memory.</td>
</tr>
</tbody>
</table>
Temporary workspace. This space may be overwritten in between calls to functions. work must be the size given in lwork.

(llocal or global) Size of user-input workspace work. If lwork is too small, the minimal acceptable size will be returned in work[0] and an error code is returned. \( lwork > (12*NPCOL+3*nb)+\max((10+2*\min(100, nrhs))*NPCOL+4*nrhs, 8*NPCOL) \).

Output Parameters

d

On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to desca[\(nb\) - 1].

e

On exit, this array contains information containing the factors of the matrix. Must be of size greater than or equal to desca[\(nb\) - 1].

b

On exit, this contains the local piece of the solutions distributed matrix \(X\).

work

On exit, work[0] contains the minimal lwork.

info

(local) If info=0, the execution is successful.

< 0: If the \(i\)-th argument is an array and the \(j\)-entry had an illegal value, then info = -(\(i*100+j\)), if the \(i\)-th argument is a scalar and had an illegal value, then info = -\(i\).

> 0: If info = \(k\) \(\leq\) NPROCS, the submatrix stored on processor info and factored locally was not positive definite, and the factorization was not completed.

If info = \(k\) > NPROCS, the submatrix stored on processor info-NPROCS representing interactions with other processors was not positive definite, and the factorization was not completed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gels

Solves overdetermined or underdetermined linear systems involving a matrix of full rank.

Syntax

void psgels (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);

void pdgels (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);

void p cgels (char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgels ( char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *nrhs , MKL_Complex16 *a ,
MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );

Include Files
- mkl_scalapack.h

Description
The pzgels function solves overdetermined or underdetermined real/complex linear systems involving an 
m-by-n matrix sub(A) = A(ia:ia+m-1,ja:ja+n-1), or its transpose/conjugate-transpose, using a QTQ or 
LQ factorization of sub(A). It is assumed that sub(A) has full rank.

The following options are provided:
1. If trans = 'N' and m≥n: find the least squares solution of an overdetermined system, that is, solve
   the least squares problem
   
   minimize ||sub(B) - sub(A)*X||

2. If trans = 'N' and m < n: find the minimum norm solution of an underdetermined system sub(A)*X = sub(B).

3. If trans = 'T' and m≥n: find the minimum norm solution of an underdetermined system sub(A)^T*X = sub(B).

4. If trans = 'T' and m < n: find the least squares solution of an overdetermined system, that is, solve
   the least squares problem
   
   minimize ||sub(B) - sub(A)^T*X||,

   where sub(B) denotes B(ib:ib+m-1, jb:jb+nrhs-1) when trans = 'N' and B(ib:ib+n-1, 
   jb:jb+nrhs-1) otherwise. Several right hand side vectors b and solution vectors x can be handled in a 
single call; when trans = 'N', the solution vectors are stored as the columns of the n-by-nrhs right 
hand side matrix sub(B) and the m-by-nrhs right hand side matrix sub(B) otherwise.

Input Parameters

trans
   (global) Must be 'N', or 'T'.
   If trans = 'N', the linear system involves matrix sub(A);
   If trans = 'T', the linear system involves the transposed matrix A^T (for 
   real flavors only).

m
   (global) The number of rows in the distributed matrix sub(A) (m≥0).

n
   (global) The number of columns in the distributed matrix sub(A) (n≥0).

nrhs
   (global) The number of right-hand sides; the number of columns in the 
distributed submatrices sub(B) and X. (nrhs≥0).

a
   (local)
   Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1). On 
   entry, contains the m-by-n matrix A.

ia, ja
   (global) The row and column indices in the global matrix A indicating the 
   first row and the first column of the submatrix A, respectively.

desca
   (global and local) array of size dlen_. The array descriptor for the 
distributed matrix A.
b

(Pointer into the local memory to an array of local size lld_b*LOCc(jb +nrhs-1). On entry, this array contains the local pieces of the distributed matrix B of right-hand side vectors, stored columnwise; sub(B) is m-by-nrhs if trans='N', and n-by-nrhs otherwise.

ib, jb

 The row and column indices in the global matrix B indicating the first row and the first column of the submatrix B, respectively.

descb

 (global and local) array of size dlen_. The array descriptor for the distributed matrix B.

work

 (local)

 Workspace array with size lwork.

lwork

 (local or global).

 The size of the array work/lwork is local input and must be at least lwork \geq ltau + \max(lwf, lws), where if m > n, then

 ltau = numroc(ja+min(m,n)-1, nb_a, MYCOL, csra_a, NPCOL),
lwf = nb_a*(mpa0 + nqa0 + nb_a)
lws = \max((nb_a*(nb_a-1))/2, (nrhsqb0 + mpb0)*nb_a) + nb_a*nb_a
else
 ltau = numroc(ia+min(m,n)-1, mb_a, MYROW, rsca_a, NPROW),
lwf = mb_a * (mpa0 + nqa0 + mb_a)
lws = \max((mb_a*(mb_a-1))/2, (npb0 + \max(nqa0 + numroc(numroc(n+iroffb, mb_a, 0, 0, NPROW), mb_a, 0, 0, lcm), nrhsqb0))*mb_a) + mb_a*mb_a
end if,

 where lcm = lcm/NPROW with lcm = ilcm(NPROW, NPCOL),
iroff = mod(ia-1, mb_a),
icoff = mod(ja-1, nb_a),
iarow = indxg2p(ia, mb_a, MYROW, rsca_a, NPROW),
iacol= indxg2p(ja, nb_a, MYROW, rsca_a, NPROW)
mpa0 = numroc(m+iroff, mb_a, MYROW, iarow, NPROW),
nqa0 = numroc(n+icoff, nb_a, MYCOL, iacol, NPCOL),
iroff = mod(ib-1, mb_b),
icoff = mod(jb-1, nb_b),
ibrow = indxg2p(ib, mb_b, MYROW, rsrb_b, NPROW),
ibcol = indxg2p(jb, nb_b, MYCOL, srcsb_b, NPCOL),
mpb0 = numroc(m+iroff, mb_b, MYROW, icrow, NPROW),
nqb0 = numroc(n+icoff, nb_b, MYCOL, ibcol, NPCOL),
NOTE
mod(x, y) is the integer remainder of \( x/y \).

ilcm, indx2p and numroc are ScaLAPACK tool functions; MYROW, MYCOL, NPROW, and NPCOL can be determined by calling the function blacs_gridinfo.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

\( a \)
On exit, If \( m \geq n \), sub(\( A \)) is overwritten by the details of its QR factorization as returned by \( p?geqrf \); if \( m < n \), sub(\( A \)) is overwritten by details of its LQ factorization as returned by \( p?gelqf \).

\( b \)
On exit, sub(\( B \)) is overwritten by the solution vectors, stored columnwise: if \( trans = 'N' \) and \( m \geq n \), rows 1 to \( n \) of sub(\( B \)) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( n+1 \) to \( m \) in that column;
If \( trans = 'N' \) and \( m < n \), rows 1 to \( n \) of sub(\( B \)) contain the minimum norm solution vectors;
If \( trans = 'T' \) and \( m \geq n \), rows 1 to \( m \) of sub(\( B \)) contain the minimum norm solution vectors; if \( trans = 'T' \) and \( m < n \), rows 1 to \( m \) of sub(\( B \)) contain the least squares solution vectors; the residual sum of squares for the solution in each column is given by the sum of squares of elements \( m+1 \) to \( n \) in that column.

\( work[0] \)
On exit, \( work[0] \) contains the minimum value of \( lwork \) required for optimum performance.

\( info \)
(global)
\( = 0 \): the execution is successful.
\( < 0 \): if the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( info = -(i*100+j) \); if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\( p?syev \)
Computes selected eigenvalues and eigenvectors of a symmetric matrix.
Syntax
void pssyev (char *jobz, char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *w, float *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *info);
void pdsyev (char *jobz, char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *w, double *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The `p?syev` function computes all eigenvalues and, optionally, eigenvectors of a real symmetric matrix $A$ by calling the recommended sequence of ScaLAPACK functions.

In its present form, the function assumes a homogeneous system and makes no checks for consistency of the eigenvalues or eigenvectors across the different processes. Because of this, it is possible that a heterogeneous system may return incorrect results without any error messages.

Input Parameters
$n_p$ = the number of rows local to a given process.
$n_q$ = the number of columns local to a given process.

`jobz` (global) Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:
If `jobz` = 'N', then only eigenvalues are computed.
If `jobz` = 'V', then eigenvalues and eigenvectors are computed.

`uplo` (global) Must be 'U' or 'L'. Specifies whether the upper or lower triangular part of the symmetric matrix $A$ is stored:
If `uplo` = 'U', $a$ stores the upper triangular part of $A$.
If `uplo` = 'L', $a$ stores the lower triangular part of $A$.

$n$ (global) The number of rows and columns of the matrix $A (n \geq 0)$.

$a$ (local)
Block cyclic array of global size $n \times n$ and local size $lld_a \times \text{LOCc}(ja+n-1)$. On entry, the symmetric matrix $A$.
If `uplo` = 'U', only the upper triangular part of $A$ is used to define the elements of the symmetric matrix.
If `uplo` = 'L', only the lower triangular part of $A$ is used to define the elements of the symmetric matrix.

`ia`, `ja` (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

desca (global and local) array of size $dlen_$. The array descriptor for the distributed matrix $A$. 
(global) The row and column indices in the global matrix $Z$ indicating the first row and the first column of the submatrix $Z$, respectively.

(global and local) array of size $dlen_$. The array descriptor for the distributed matrix $Z$.

(local) Array of size $lwork$.

(local) See below for definitions of variables used to define $lwork$.

If no eigenvectors are requested ($jobz = 'N'$), then $lwork \geq 5*n + $ sizesytrd + 1, where sizesytrd is the workspace for $p?sytrd$ and is $\max(NB*(np +1)$, $3*NB$).

If eigenvectors are requested ($jobz = 'V'$) then the amount of workspace required to guarantee that all eigenvectors are computed is:

$$qrmem = 2*n-2$$

$$lwmin = 5*n + n*ldc + \max(sizemqrleft, qrmem) + 1$$

Variable definitions:


$$nn = \max(n, nb, 2);$$


$$np = \text{numroc}(nn, nb, 0, 0, NPROW)$$

$$nq = \text{numroc}(\max(n, nb, 2), nb, 0, 0, NPCOL)$$

$$nrc = \text{numroc}(n, nb, myprowc, 0, NPROCS)$$

$$ldc = \max(1, nrc)$$

$sizemqrleft$ is the workspace for $p?ormtr$ when its side argument is 'L'.

$myprowc$ is defined when a new context is created as follows:

```
call blacs_get(desca[ctxt - 1], 0, contextc)
call blacs_gridinit(contextc, 'R', NPROCS, 1)
call blacs_gridinfo(contextc, nprowc, npcolc, myprowc, mypcolc)
```

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by $pxerbla$.

### Output Parameters

$a$

On exit, the lower triangle (if $uplo='L'$) or the upper triangle (if $uplo='U'$) of $A$, including the diagonal, is destroyed.

$w$

(global).
Array of size $n$.
On normal exit, the first $m$ entries contain the selected eigenvalues in ascending order.

$z$
Array, global size $n \times n$, local size $lld_z \times LOCc(jz+n-1)$. If $jobz = 'V$', then on normal exit the first $m$ columns of $z$ contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.
If $jobz = 'N'$, then $z$ is not referenced.

work[0]
On output, $work[0]$ returns the workspace needed to guarantee completion. If the input parameters are incorrect, $work[0]$ may also be incorrect.
If $jobz = 'N'$, $work[0] = \text{minimal (optimal) amount of workspace}$
If $jobz = 'V'$, $work[0] = \text{minimal workspace required to generate all the eigenvectors}$

info (global)
If $info = 0$, the execution is successful.
If $info < 0$: If the $i$-th argument is an array and the $j$-entry had an illegal value, then $info = -(i \times 100 + j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.
If $info > 0$:
If $info = 1$ through $n$, the $i$-th eigenvalue did not converge in steqr2 after a total of $30n$ iterations.
If $info = n+1$, then p?syev has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from p?syev cannot be guaranteed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syevd
Computes all eigenvalues and eigenvectors of a real symmetric matrix by using a divide and conquer algorithm.

Syntax
void pssyevd (char *jobz , char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *w , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
void pdsyevd (char *jobz , char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *w , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *liwork , MKL_INT *info );
Include Files

- mkl_scalapack.h

Description

The `p?syevd` function computes all eigenvalues and eigenvectors of a real symmetric matrix `A` by using a divide and conquer algorithm.

Input Parameters

`np` = the number of rows local to a given process.

`nq` = the number of columns local to a given process.

`jobz` (global) Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If `jobz` = 'N', then only eigenvalues are computed.

If `jobz` = 'V', then eigenvalues and eigenvectors are computed.

`uplo` (global) Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix `A` is stored:

If `uplo` = 'U', `a` stores the upper triangular part of `A`.

If `uplo` = 'L', `a` stores the lower triangular part of `A`.

`n` (global) The number of rows and columns of the matrix `A` (`n ≥ 0`).

`a` (local).

Block cyclic array of global size `n*n` and local size `lld_a*LOCc(ja+n-1)`. On entry, the symmetric matrix `A`.

If `uplo` = 'U', only the upper triangular part of `A` is used to define the elements of the symmetric matrix.

If `uplo` = 'L', only the lower triangular part of `A` is used to define the elements of the symmetric matrix.

`ia`, `ja` (global) The row and column indices in the global matrix `A` indicating the first row and the first column of the submatrix `A`, respectively.

`desca` (global and local) array of size `dlenn_.` The array descriptor for the distributed matrix `A`. If `desca[ctxt_ - 1]` is incorrect, `p?syevd` cannot guarantee correct error reporting.

`iz`, `jz` (global) The row and column indices in the global matrix `Z` indicating the first row and the first column of the submatrix `Z`, respectively.

`descz` (global and local) array of size `dlenn_.` The array descriptor for the distributed matrix `Z`. `descz[ctxt_ - 1]` must equal `desca[ctxt_ - 1]`.

`work` (local).

Array of size `lwork`.

`lwork` (local) The size of the array `work`.

If eigenvalues are requested:
lwork ≥ \( \max(1 + 6n + 2npnq, \text{trilwmin}) + 2n \)
with \( \text{trilwmin} = 3n + \max(nb(n + 1), 3nb) \)

\( np = \text{numroc}(n, nb, \text{myrow}, \text{iarow}, \text{NPROW}) \)
\( nq = \text{numroc}(n, nb, \text{mycol}, \text{iacol}, \text{NPCOL}) \)

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

\( iwork \)
(local) Workspace array of size \( liwork \).
\( liwork \)
(local), size of \( iwork \).
\( liwork = 7n + 8nq + 2 \).

Output Parameters

\( a \)
On exit, the lower triangle (if \( \text{uplo} = 'L' \)) or the upper triangle (if \( \text{uplo} = 'U' \)) of \( A \), including the diagonal, is overwritten.

\( w \)
Array of size \( n \). If \( info = 0 \), \( w \) contains the eigenvalues in the ascending order.

\( z \)
(local).
Array, global size \((n, n)\), local size \( lld_z*\text{LOCc}(jz+n-1) \).
The \( z \) parameter contains the orthonormal eigenvectors of the matrix \( A \).

\( \text{work}[0] \)
On exit, returns adequate workspace to allow optimal performance.

\( \text{iwork}[0] \)
On exit, if \( liwork > 0 \), \( \text{iwork}[0] \) returns the optimal \( liwork \).

\( info \)
(global)
If \( info = 0 \), the execution is successful.
If \( info < 0 \):
If the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( info = -(i*100+j) \). If the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).
If \( info > 0 \):
The algorithm failed to compute the \( info/(n+1) \)-th eigenvalue while working on the submatrix lying in global rows and columns \( \text{mod}(info, n + 1) \).

**NOTE**
\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?syevr
Computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix using Relatively Robust Representation.

Syntax
void pssyevr(char* jobz, char* range, char* uplo, MKL_INT* n, float* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, MKL_INT* nz, float* w, float* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);

void pdsyevr(char* jobz, char* range, char* uplo, MKL_INT* n, double* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, MKL_INT* nz, double* w, double* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);

Include Files
- mkl_scalapack.h

Description
p?syevr computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK functions.

First, the matrix A is reduced to real symmetric tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in w. The eigenvector matrix z is stored in 2D block-cyclic format distributed over all processors.

Note that subsets of eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Input Parameters

jobz (global)
Specifies whether or not to compute the eigenvectors:
= 'N': Compute eigenvalues only.
= 'V': Compute eigenvalues and eigenvectors.
**range** (global)

- 'A': all eigenvalues will be found.
- 'V': all eigenvalues in the interval \([vl, vu]\) will be found.
- 'I': the \(i_l\)-th through \(i_u\)-th eigenvalues will be found.

**uplo** (global)

Specifies whether the upper or lower triangular part of the symmetric matrix \(A\) is stored:
- 'U': Upper triangular
- 'L': Lower triangular

**n** (global)

The number of rows and columns of the matrix \(A\). \(n \geq 0\)

**a**

Block cyclic array of global size \(n \times n\), local size \(lld_a \times LOC_c(ja+n-1)\).

This array contains the local pieces of the symmetric distributed matrix \(A\). If \(uplo = 'U'\), only the upper triangular part of \(a\) is used to define the elements of the symmetric matrix. If \(uplo = 'L'\), only the lower triangular part of \(a\) is used to define the elements of the symmetric matrix.

On exit, the lower triangle (if \(uplo='L'\)) or the upper triangle (if \(uplo='U'\)) of \(a\), including the diagonal, is destroyed.

**ia** (global)

Global row index in the global matrix \(A\) that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

**ja** (global)

Global column index in the global matrix \(A\) that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

**desca** (global and local) array of size \(dlen_\_=9\).

The array descriptor for the distributed matrix \(a\).

**vl** (global)

If \(range='V'\), the lower bound of the interval to be searched for eigenvalues. Not referenced if \(range = 'A'\) or 'I'.

**vu** (global)

If \(range='V'\), the upper bound of the interval to be searched for eigenvalues. Not referenced if \(range = 'A'\) or 'I'.

**il** (global)

If \(range='I'\), the index (from smallest to largest) of the smallest eigenvalue to be returned. \(il \geq 1\).

Not referenced if \(range = 'A'\).

**iu** (global)
If range='I', the index (from smallest to largest) of the largest eigenvalue to be returned. min(il,n) ≤ iu ≤ n.

Not referenced if range = 'A'.

iz

(global )

Global row index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

jz

(global )

Global column index in the global matrix Z that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

descz

array of size dlen_.

The array descriptor for the distributed matrix z.

The context descz[ctxt_ - 1] must equal desca[ctxt_ - 1]. Also note the array alignment requirements specified below.

work

(local workspace) array of size lwork

lwork

(local )

Size of work, must be at least 3.

See below for definitions of variables used to define lwork.

If no eigenvectors are requested (jobz = 'N') then

lwork≥ 2 + 5*n + max( 12 * nn, neig * ( np0 + 1 ) )

If eigenvectors are requested (jobz = 'V' ) then the amount of workspace required is:

lwork≥ 2 + 5*n + max( 18*nn, np0 * mq0 + 2 * neig * neig ) + (2 + iceil( neig, nprow*ncol))*nn

Variable definitions:

neig = number of eigenvectors requested

nb = desca[ mb_ - 1] = desca( nb_ ) = descz[ mb_ - 1] = descz( nb_ )

nn = max( n, neig, 2 )


np0 = numroc( nn, neig, 0, 0, nprow )

mq0 = numroc( max( neig, neig, 2 ), neig, 0, 0, npcol )

iceil( x, y ) is a ScALAPACK function returning ceiling(x/y), and nprow and npcol can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by pxerbla.
liwork

(size of iwork)

Let \( nnp = \max(n, nprow \times npcol + 1, 4) \). Then:

\[
liwork \geq 12 \times nnp + 2 \times n \quad \text{when the eigenvectors are desired}
\]

\[
liwork \geq 10 \times nnp + 2 \times n \quad \text{when only the eigenvalues have to be computed}
\]

If \( liwork = -1 \), then \( liwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \texttt{pxerbla}.

**OUTPUT Parameters**

\( m \) (global )

Total number of eigenvalues found. 0 \( \leq m \leq n \).

\( nz \) (global )

Total number of eigenvectors computed. 0 \( \leq nz \leq m \).

The number of columns of \( z \) that are filled.

If \( jobz \neq 'V' \), \( nz \) is not referenced.

If \( jobz = 'V' \), \( nz = m \)

\( w \) (global ) array of size \( n \)

Upon successful exit, the first \( m \) entries contain the selected eigenvalues in ascending order.

\( z \)

Block-cyclic array, global size \( n \times n \), local size \( lld_z \times LOC_C(jz+n-1) \).

On exit, contains local pieces of distributed matrix \( Z \).

\( work \)

On return, \( work[0] \) contains the optimal amount of workspace required for efficient execution. If \( jobz = 'N' \), \( work[0] \) = optimal amount of workspace required to compute the eigenvalues. If \( jobz = 'V' \), \( work[0] \) = optimal amount of workspace required to compute eigenvalues and eigenvectors.

\( iwork \) (local workspace) array

On return, \( iwork[0] \) contains the amount of integer workspace required.

\( info \) (global )

= 0: successful exit

< 0: If the \( i \)-th argument is an array and the \( j \)-th entry had an illegal value, then \( info = -(i \times 100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

**Application Notes**

The distributed submatrices \( a(ia:*, ja:* \) and \( z(iz:iz+m-1, jz:jz+n-1) \) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: \( desca[m - 1] = descz[m - 1] = desca[n - 1] = descz[n - 1] \)
2. Quadratic conformal blocking: \( \text{desca}[mb - 1] = \text{desca}[nb - 1] = \text{descz}[mb - 1] = \text{descz}[nb - 1] \),
\( \text{desca}[src - 1] = \text{descz}[src - 1] \)

3. \( \text{mod}( ia - 1, mb_a ) = \text{mod}( iz - 1, mb_z ) = 0 \)

**NOTE**
\( \text{mod}(x, y) \) is the integer remainder of \( x/y \).

See Also
Overview for details of ScalAPACK array descriptor structures and related notations.

**p?syevx**
Computes selected eigenvalues and, optionally, eigenvectors of a symmetric matrix.

**Syntax**
```c
void pssyevx (char *jobz, char *range, char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *vl, float *vu, MKL_INT *il, MKL_INT *iu, float *abstol, MKL_INT *m, MKL_INT *nz, float *w, float *orfac, float *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, float *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *iifail, MKL_INT *iclustr, float *gap, MKL_INT *info );
void pdsyevx (char *jobz, char *range, char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *vl, double *vu, MKL_INT *il, MKL_INT *iu, double *abstol, MKL_INT *m, MKL_INT *nz, double *w, double *orfac, double *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, double *work, MKL_INT *lwork, MKL_INT *iwork, MKL_INT *iifail, MKL_INT *iclustr, double *gap, MKL_INT *info );
```

**Include Files**
- mkl_scalapack.h

**Description**
The p?syevx function computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix \( A \) by calling the recommended sequence of ScalAPACK functions. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

**Optimization Notice**
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**
\( np \) = the number of rows local to a given process.
$nq$ = the number of columns local to a given process.

$\text{jobz}$ (global) Must be 'N' or 'V'. Specifies if it is necessary to compute the eigenvectors:

- If $\text{jobz} =$'N', then only eigenvalues are computed.
- If $\text{jobz} =$'V', then eigenvalues and eigenvectors are computed.

$\text{range}$ (global) Must be 'A', 'V', or 'I'.

- If $\text{range} =$ 'A', all eigenvalues will be found.
- If $\text{range} =$ 'V', all eigenvalues in the half-open interval $[vl, vu]$ will be found.
- If $\text{range} =$ 'I', the eigenvalues with indices $il$ through $iu$ will be found.

$\text{uplo}$ (global) Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the symmetric matrix $A$ is stored:

- If $\text{uplo} =$ 'U', $a$ stores the upper triangular part of $A$.
- If $\text{uplo} =$ 'L', $a$ stores the lower triangular part of $A$.

$n$ (global) The number of rows and columns of the matrix $A$ ($n \geq 0$).

$a$ (local).

Block cyclic array of global size $n \times n$ and local size $lld_a \times \text{LOCc}(ja+n-1)$.

On entry, the symmetric matrix $A$.

- If $\text{uplo} =$ 'U', only the upper triangular part of $A$ is used to define the elements of the symmetric matrix.
- If $\text{uplo} =$ 'L', only the lower triangular part of $A$ is used to define the elements of the symmetric matrix.

$\text{ia, ja}$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

$\text{desca}$ (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$vl, vu$ (global)

- If $\text{range} =$ 'V', the lower and upper bounds of the interval to be searched for eigenvalues; $vl \leq vu$. Not referenced if $\text{range} =$ 'A' or 'I'.

$il, iu$ (global)

- If $\text{range} =$'I', the indices of the smallest and largest eigenvalues to be returned.

Constraints: $il \geq 1$

$\min(il, n) \leq iu \leq n$

Not referenced if $\text{range} =$ 'A' or 'V'.

$\text{abstol}$ (global).

- If $\text{jobz} =$'V', setting $\text{abstol}$ to $\text{p?lamch(context, 'U')}$ yields the most orthogonal eigenvectors.
The absolute error tolerance for the eigenvalues. An approximate
eigenvalue is accepted as converged when it is determined to lie in an
interval \([a, b]\) of width less than or equal to

\[abstol + eps \times \max(|a|, |b|),\]

where \(eps\) is the machine precision. If \(abstol\) is less than or equal to zero,
then \(eps\times\text{norm}(T)\) will be used in its place, where \(\text{norm}(T)\) is the 1-norm of
the tridiagonal matrix obtained by reducing \(A\) to tridiagonal form.

Eigenvalues will be computed most accurately when \(abstol\) is set to twice
the underflow threshold \(2\times\text{lamch}'S'\) not zero. If this function returns
with \((\text{mod(info,2)} \neq 0)\) or \((\text{mod(info/8,2)} \neq 0)\), indicating that some
eigenvalues or eigenvectors did not converge, try setting \(abstol\) to
\(2\times\text{lamch}'S'\).

\(orfac\)

(global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that
correspond to eigenvalues which are within \(tol=orfac\times\text{norm}(A)\) of each
other are to be reorthogonalized. However, if the workspace is insufficient
(see \(lwork\)), \(tol\) may be decreased until all eigenvectors to be
reorthogonalized can be stored in one process. No reorthogonalization will
be done if \(orfac\) equals zero. A default value of 1.0e-3 is used if \(orfac\) is
negative. \(orfac\) should be identical on all processes.

\(iz, jz\)

(global) The row and column indices in the global matrix \(Z\) indicating the
first row and the first column of the submatrix \(Z\), respectively.

\(descz\)

(global and local) array of size \(dlen_\).
The array descriptor for the
distributed matrix \(Z\). \(descz[ctxt_ - 1]\) must equal \(desca[ctxt_ - 1]\).

\(work\)

(local)

Array of size \(lwork\).

\(lwork\)

(local) The size of the array \(work\).

See below for definitions of variables used to define \(lwork\).

If no eigenvectors are requested \((jobz = 'N')\), then
\(lwork \geq 5\times n + \max(5\times n, \text{NB} \times (np0 + 1))\).

If eigenvectors are requested \((jobz = 'V')\), then the amount of workspace
required to guarantee that all eigenvectors are computed is:

\(lwork \geq 5\times n + \max(5\times n, np0 \times mq0 + 2 \times \text{NB} \times \text{NB}) + \text{iceil}(\text{neig}, \text{NPROW} \times \text{NPCOL}) \times n\)

The computed eigenvectors may not be orthogonal if the minimal
workspace is supplied and \(orfac\) is too small. If you want to guarantee
orthogonality (at the cost of potentially poor performance) you should add
the following to \(lwork\):

\((\text{clustersize}-1) \times n,\)

where \(\text{clustersize}\) is the number of eigenvalues in the largest cluster, where
a cluster is defined as a set of close eigenvalues:

\([w[k - 1], ..., w[k+\text{clustersize}-2]|w[j] \leq w[j-1]) + orfac \times 2 \times \text{norm}(A),\)
where

\( \text{neig} = \text{number of eigenvectors requested} \)
\( \text{nb} = \text{desca}[\text{mb} - 1] = \text{desca}[\text{nb} - 1] = \text{descz}[\text{mb} - 1] = \text{descz}[\text{nb} - 1]; \)
\( \text{nn} = \max(n, \text{nb}, 2); \)
\( \text{desca}[\text{rsr}_c - 1] = \text{desca}[\text{nb} - 1] = \text{descz}[\text{rsr}_c - 1] = \text{descz}[\text{csr}_c - 1] = 0; \)
\( np0 = \text{numroc}(n, \text{nb}, 0, 0, \text{NPROW}); \)
\( mq0 = \text{numroc}(\max(\text{neig}, \text{nb}, 2), \text{nb}, 0, 0, \text{NPCOL}); \)
\( \text{iceil}(x, y) \text{ is a ScaLAPACK function returning ceiling}(x/y) \)

If \( lwork \) is too small to guarantee orthogonality, \( \text{p?syevx} \) attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If \( lwork \) is too small to compute all the eigenvectors requested, no computation is performed and \( \text{info} = -23 \) is returned.

Note that when \( \text{range} = 'V' \), number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if \( lwork \) is large enough to compute the eigenvalues, \( \text{p?sygvx} \) computes the eigenvalues and as many eigenvectors as possible.

\textbf{Relationship between workspace, orthogonality & performance:}

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

\[ lwork \geq \max(lwork, 5*n + n\text{symrd}_{lwopt}) \]

where \( lwork \), as defined previously, depends upon the number of eigenvectors requested, and

\[ n\text{sytrd}_{lwopt} = n + 2*(anb+1)*(4*nps+2) + (nps + 3)*nps; \]
\[ anb = \text{pjlaenv}(\text{desca}[\text{ctxt} - 1], 3, 'p\text{?sytrd}', 'L', 0, 0, 0, 0); \]
\[ \text{sqnpc} = \text{int}(\text{sqrt}(\text{dble}(\text{NPROW} * \text{NPCOL}))); \]
\[ nps = \max(\text{numroc}(n, 1, 0, 0, \text{sqnpc}), 2*anb); \]
\[ \text{numroc} \text{ is a ScaLAPACK tool functions;} \]
\[ \text{pjlaenv} \text{ is a ScaLAPACK environmental inquiry function} \]

\( \text{MYROW, MYCOL, NPROW and NPCOL} \) can be determined by calling the function \( \text{blacs_gridinfo} \).

For large \( n \), no extra workspace is needed, however the biggest boost in performance comes for small \( n \), so it is wise to provide the extra workspace (typically less than a megabyte per process).

If \( \text{clustersize} > n/\text{sqrt}(\text{NPROW}\times\text{NPCOL}) \), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, \( \text{clustersize} = n-1 \) \( \text{p?stein} \) will perform no better than \( ?\text{stein} \) on single processor.)
For \( \text{clustersize} = n / \sqrt{NPROW \times NPCOL} \) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For \( \text{clustersize} > n / \sqrt{NPROW \times NPCOL} \) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \text{pxerbla}.

\text{iwork} \\
(\text{local}) \text{ Workspace array.}

\text{liwork} \\
(\text{local}), \text{ size of iwork}. \text{liwork} \geq 6 \times \text{nnp}

Where: \( \text{nnp} = \max(n, NPROW \times NPCOL + 1, 4) \)

If \( liwork = -1 \), then \( liwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\text{a} \\
On exit, the lower triangle (if uplo = 'L') or the upper triangle (if uplo = 'U') of \( A \), including the diagonal, is overwritten.

\text{m} \\
(globally) The total number of eigenvalues found; \( 0 \leq m \leq n \).

\text{nz} \\
(globally) The total number of eigenvectors computed. \( 0 \leq nz \leq m \).

The number of columns of \( z \) that are filled.

If \( \text{jobz} \neq 'V' \), \( nz \) is not referenced.

If \( \text{jobz} = 'V' \), \( nz = m \) unless the user supplies insufficient space and \text{p?syevx} is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in \( z \) \((m \leq \text{descz}[n-1])\) and sufficient workspace to compute them. (See \text{lwork}). \text{p?syevx} is always able to detect insufficient space without computation unless \text{range} = 'V'.

\text{w} \\
(\text{global}).

Array of size \( n \). The first \( m \) elements contain the selected eigenvalues in ascending order.

\text{z} \\
(\text{local}).

Array, global size \( n \times n \), local size \( lld_z \times \text{LOCc}(jz+n-1) \).

If \( \text{jobz} = 'V' \), then on normal exit the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \text{ifail}. 

1470
If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

\text{work}[0]

On exit, returns workspace adequate workspace to allow optimal performance.

\text{iwork}[0]

On return, \( \text{iwork}[0] \) contains the amount of integer workspace required.

\text{ifail}

Array of size \( n \).

If \( \text{jobz} = 'V' \), then on normal exit, the first \( m \) elements of \( \text{ifail} \) are zero. If \( \mod(\text{info},2) \neq 0 \) on exit, then \( \text{ifail} \) contains the indices of the eigenvectors that failed to converge.

If \( \text{jobz} = 'N' \), then \( \text{ifail} \) is not referenced.

\text{iclustr}

(global) Array of size \((2*\text{NPROW}*\text{NPCOL})\)

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see \text{iwork}, \text{orfac} and \text{info}). Eigenvectors corresponding to clusters of eigenvalues indexed \( \text{iclustr}(2*i-1) \) to \( \text{iclustr}(2*i) \), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. \text{iclustr} is a zero terminated array, \text{iclustr}[2*k - 1] \neq 0 \text{ and } \text{iclustr}[2*k] = 0 \text{ if and only if } k \text{ is the number of clusters.}

\text{iclustr} \text{ is not referenced if } \text{jobz} = 'N'.

\text{gap}

(global)

Array of size \( \text{NPROW}*\text{NPCOL} \)

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array \text{iclustr}. As a result, the dot product between eigenvectors corresponding to the \( i \)th cluster may be as high as \( (C*n)/\text{gap}[i - 1] \) where \( C \) is a small constant.

\text{info}

(global)

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} < 0 \):

If the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( \text{info} = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

If \( \text{info} > 0 \):

if \( \mod(\text{info},2)\neq0 \), then one or more eigenvectors failed to converge. Their indices are stored in \text{ifail}. Ensure \( \text{abstol}=2.0*\text{p?lamch('U')} \).

If \( \mod(\text{info}/2,2)\neq0 \), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array \text{iclustr}.

If \( \mod(\text{info}/4,2)\neq0 \), then space limit prevented \( \text{p?syevxf} \) computing all of the eigenvectors between \( vl \) and \( vu \). The number of eigenvectors computed is returned in \( nz \).
If \( \text{mod}(\text{info}/8,2)\neq 0 \), then \( \text{p?stebz} \) failed to compute eigenvalues. Ensure \( \text{abstol}=2.0*\text{p?lamch('U')} \).

NOTE
\[ \text{mod}(x,y) \] is the integer remainder of \( x/y \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?heev**
Computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix.

**Syntax**

```c
void pcheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *w , MKL_Complex8 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex8 *work , MKL_INT *lwork , float *rwork , MKL_INT *lrwork , MKL_INT *info);
void pzheev (char *jobz , char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *w , MKL_Complex16 *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , MKL_Complex16 *work , MKL_INT *lwork , double *rwork , MKL_INT *lrwork , MKL_INT *info);
```

**Include Files**
* mkl_scalapack.h

**Description**
The p?heev function computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix \( A \) by calling the recommended sequence of ScaLAPACK functions. The function assumes a homogeneous system and makes spot checks of the consistency of the eigenvalues across the different processes. A heterogeneous system may return incorrect results without any error messages.

**Input Parameters**

\( np \) = the number of rows local to a given process.

\( nq \) = the number of columns local to a given process.

\( \text{jobz} \)
(global) Must be 'N' or 'V'.

Specifies if it is necessary to compute the eigenvectors:

If \( \text{jobz} = 'N' \), then only eigenvalues are computed.

If \( \text{jobz} = 'V' \), then eigenvalues and eigenvectors are computed.

\( \text{uplo} \)
(global) Must be 'U' or 'L'.

Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is stored:

If \( \text{uplo} = 'U' \), \( a \) stores the upper triangular part of \( A \).

If \( \text{uplo} = 'L' \), \( a \) stores the lower triangular part of \( A \).
The number of rows and columns of the matrix $A$ ($n \geq 0$).

Block cyclic array of global size $n \times n$ and local size $lld_a \times \text{LOC}(ja+n-1)$. On entry, the Hermitian matrix $A$.

If $\text{uplo} = 'U'$, only the upper triangular part of $A$ is used to define the elements of the Hermitian matrix.

If $\text{uplo} = 'L'$, only the lower triangular part of $A$ is used to define the elements of the Hermitian matrix.

The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

(global and local) array of size $\text{dlen}_\text{a}$. The array descriptor for the distributed matrix $A$. If $\text{desca}[\text{ctxt} - 1]$ is incorrect, $p\text{?heev}$ cannot guarantee correct error reporting.

The row and column indices in the global matrix $Z$ indicating the first row and the first column of the submatrix $Z$, respectively.

(global and local) array of size $\text{dlen}_\text{z}$. The array descriptor for the distributed matrix $Z$. $\text{descz}[\text{ctxt} - 1]$ must equal $\text{desca}[\text{ctxt} - 1]$.

Array of size $lwork$.

The size of the array $\text{work}$.

If only eigenvalues are requested ($\text{jobz} = 'N'$):

$lwork \geq \max(nb \times np0 + 1, 3) + 3 \times n$

If eigenvectors are requested ($\text{jobz} = 'V'$), then the amount of workspace required:

$lwork \geq (np0 + nq0 + nb) \times nb + 3 \times n + n^2$

with $nb = \text{desca}[mb - 1] = \text{desca}[nb - 1] = nb = \text{descz}[mb - 1] = \text{descz}[nb - 1]$

$np0 = \text{numroc}(nn, nb, 0, 0, \text{NPROW})$.

$nq0 = \text{numroc}(\max(n, nb, 2), nb, 0, 0, \text{NPCOL})$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by $\text{pxerbla}$.

Workspace array of size $lrwork$.

The size of the array $\text{rwork}$.

See below for definitions of variables used to define $lrwork$.

If no eigenvectors are requested ($\text{jobz} = 'N'$), then $lrwork \geq 2 \times n$.

If eigenvectors are requested ($\text{jobz} = 'V'$), then $lrwork \geq 2 \times n + 2 \times n - 2$. 

1473
If \( lrwork = -1 \), then \( lrwork \) is global input and a workspace query is assumed; the function only calculates the minimum size required for the \( rwork \) array. The required workspace is returned as the first element of \( rwork \), and no error message is issued by \texttt{pxerbla}.

**Output Parameters**

\textbf{a}

On exit, the lower triangle (if \( uplo = 'L' \)), or the upper triangle (if \( uplo = 'U' \)) of \( A \), including the diagonal, is overwritten. (global).

\textbf{w}

Array of size \( n \). The first \( m \) elements contain the selected eigenvalues in ascending order. (local).

\textbf{z}

Array, global size \( n \times n \), local size \( lld_z \times \text{LOCc}(jz+n-1) \).

If \( jobz = 'V' \), then on normal exit the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in \( ifail \).

If \( jobz = 'N' \), then \( z \) is not referenced.

\textbf{work[0]}

On exit, returns adequate workspace to allow optimal performance. If \( jobz = 'N' \), then \( \text{work[0]} = \) minimal workspace only for eigenvalues. If \( jobz = 'V' \), then \( \text{work[0]} = \) minimal workspace required to generate all the eigenvectors. (local)

\textbf{rwork[0]}

On output, \( rwork[0] \) returns workspace required to guarantee completion. (local)

\textbf{info}

(global)

If \( info = 0 \), the execution is successful.

If \( info < 0 \):

If the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( info = -(i*100+j) \). If the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

If \( info > 0 \):

If \( info = 1 \) through \( n \), the \( i \)-th eigenvalue did not converge in \?steqr2 after a total of \( 30 \times n \) iterations.

If \( info = n+1 \), then \texttt{p?heev} detected heterogeneity, and the accuracy of the results cannot be guaranteed.

**See Also**

**Overview** for details of ScALAPACK array descriptor structures and related notations.
Computes all eigenvalues and eigenvectors of a complex Hermitian matrix by using a divide and conquer algorithm.

Syntax

```c
void pcheevd (char *jobz, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *w, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *lrwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);
void pzheevd (char *jobz, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *w, MKL_Complex16 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *lrwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *info);
```

Include Files

- mkl_scalapack.h

Description

The `p?heevd` function computes all eigenvalues and eigenvectors of a complex Hermitian matrix \( A \) by using a divide and conquer algorithm.

Input Parameters

- \( np \) = the number of rows local to a given process.
- \( nq \) = the number of columns local to a given process.

\( \text{jobz} \) (global) Must be 'N' or 'V'.

- Specifies if it is necessary to compute the eigenvectors:
  - If \( \text{jobz} = 'N' \), then only eigenvalues are computed.
  - If \( \text{jobz} = 'V' \), then eigenvalues and eigenvectors are computed.

\( \text{uplo} \) (global) Must be 'U' or 'L'.

- Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is stored:
  - If \( \text{uplo} = 'U' \), \( a \) stores the upper triangular part of \( A \).
  - If \( \text{uplo} = 'L' \), \( a \) stores the lower triangular part of \( A \).

\( n \) (global) The number of rows and columns of the matrix \( A (n \geq 0) \).

\( a \) (local).

- Block cyclic array of global size \( n \times n \) and local size \( lld_a * \text{LOCc}(ja+n-1) \).
- On entry, the Hermitian matrix \( A \).
- If \( \text{uplo} = 'U' \), only the upper triangular part of \( A \) is used to define the elements of the Hermitian matrix.
- If \( \text{uplo} = 'L' \), only the lower triangular part of \( A \) is used to define the elements of the Hermitian matrix.
ia, ja  
(global) The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca  
(global and local) array of size dlen_. The array descriptor for the distributed matrix A. If desca[ctxt_ - 1] is incorrect, p?heevd cannot guarantee correct error reporting.

iz, jz  
(global) The row and column indices in the global matrix Z indicating the first row and the first column of the submatrix Z, respectively.

descz  
(global and local) array of size dlen_. The array descriptor for the distributed matrix Z. descz[ctxt_ - 1] must equal desca[ctxt_ - 1].

work  
(local).
Array of size lwork.
lwork  
(local) The size of the array work.
If eigenvalues are requested:
lwork = n + (nb0 + mq0 + nb)*nb
with np0 = numroc( max( n, nb, 2 ), nb, 0, 0, NPROW)
mq0 = numroc( max( n, nb, 2 ), nb, 0, 0, NPCOL)
If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. The required workspace is returned as the first element of the corresponding work arrays, and no error message is issued by pxerbla.

rwork  
(local).
Workspace array of size lrwork.
lrwork  
(local) The size of the array rwork.
lrwork ≥ 1 + 9*n + 3*np*nq,
with np = numroc( n, nb, myrow, iarow, NPROW)
nq = numroc( n, nb, mycol, iacol, NPCOL)
iwork  
(local) Workspace array of size liwork.
liwork  
(local), size of iwork.
liwork = 7*n + 8*npcol + 2.

Output Parameters

a  
On exit, the lower triangle (if uplo = 'L'), or the upper triangle (if uplo = 'U') of A, including the diagonal, is overwritten.

w  
(global).
Array of size n. If info = 0, w contains the eigenvalues in the ascending order.

z  
(local).
Array, global size n*n, local size lld_z*LOCc(jz+n-1).
The \( z \) parameter contains the orthonormal eigenvectors of the matrix \( A \).

\textbf{work[0]}  
On exit, returns adequate workspace to allow optimal performance.

\textbf{rwork[0]}  
(local)  
On output, \textit{rwork[0]} returns workspace required to guarantee completion.

\textbf{iwork[0]}  
(local)  
On return, \textit{iwork[0]} contains the amount of integer workspace required.

\textbf{info}  
(global)  
If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} < 0 \):
If the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( \text{info} = -(i*100+j) \). If the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

If \( \text{info} > 0 \):
If \( \text{info} = 1 \) through \( n \), the \( i \)-th eigenvalue did not converge.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?heevr}

\textit{Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix using Relatively Robust Representation.}

\textbf{Syntax}

\begin{verbatim}
void pcheevr(char* jobz, char* range, char* uplo, MKL_INT* n, MKL_Complex8* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, MKL_INT* nz, float* w, MKL_Complex8* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, MKL_Complex8* work, MKL_INT* lwork, float* rwork, MKL_INT* lrwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
void pzheevr(char* jobz, char* range, char* uplo, MKL_INT* n, MKL_Complex16* a, MKL_INT* ia, MKL_INT* ja, MKL_INT* desca, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, MKL_INT* nz, double* w, MKL_Complex16* z, MKL_INT* iz, MKL_INT* jz, MKL_INT* descz, MKL_Complex16* work, MKL_INT* lwork, double* rwork, MKL_INT* lrwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
\end{verbatim}

\textbf{Include Files}

- mkl_scalapack.h

\textbf{Description}

\textit{p?heevr} computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix \( A \) distributed in 2D blockcyclic format by calling the recommended sequence of ScaLAPACK functions.

First, the matrix \( A \) is reduced to complex Hermitian tridiagonal form. Then, the eigenproblem is solved using the parallel MRRR algorithm. Last, if eigenvectors have been computed, a backtransformation is done.

Upon successful completion, each processor stores a copy of all computed eigenvalues in \( w \). The eigenvector matrix \( Z \) is stored in 2D block-cyclic format distributed over all processors.
Input Parameters

**jobz**

Specifies whether or not to compute the eigenvectors:

- 'N': Compute eigenvalues only.
- 'V': Compute eigenvalues and eigenvectors.

**range**

Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is stored:

- 'U': Upper triangular
- 'L': Lower triangular

**n**

The number of rows and columns of the matrix \( A \). \( n \geq 0 \)

**a**

Block-cyclic array, global size \( n * n \), local size \( lld_a * LOC_c(ja+n-1) \)

Contains the local pieces of the Hermitian distributed matrix \( A \). If \( uplo = 'U' \), only the upper triangular part of \( a \) is used to define the elements of the Hermitian matrix. If \( uplo = 'L' \), only the lower triangular part of \( a \) is used to define the elements of the Hermitian matrix.

**ia**

Global row index in the global matrix \( A \) that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.

**ja**

Global column index in the global matrix \( A \) that points to the beginning of the submatrix which is to be operated on. It should be set to 1 when operating on a full matrix.
desca  (global and local) array of size dlen_. (The ScaLAPACK descriptor length is 
dlen_ = 9.)

The array descriptor for the distributed matrix a. The descriptor stores 
details about the 2D block-cyclic storage, see the notes below. If desca is 
incorrect, p?heevr cannot work correctly.

Also note the array alignment requirements specified below

vl  (global)

If range='V', the lower bound of the interval to be searched for 
eigenvalues. Not referenced if range = 'A' or 'I'.

vu  (global)

If range='V', the upper bound of the interval to be searched for 
eigenvalues. Not referenced if range = 'A' or 'I'.

il  (global )

If range='I', the index (from smallest to largest) of the smallest eigenvalue 
to be returned. il≥ 1.

Not referenced if range = 'A'.

iu  (global )

If range='I', the index (from smallest to largest) of the largest eigenvalue 
to be returned. min(il,n)≤iu≤n.

Not referenced if range = 'A'.

iz  (global )

Global row index in the global matrix Z that points to the beginning of the 
submatrix which is to be operated on. It should be set to 1 when operating 
on a full matrix.

jz  (global )

Global column index in the global matrix Z that points to the beginning of the 
submatrix which is to be operated on. It should be set to 1 when 
operating on a full matrix.

descz  (global and local) array of size dlen_.

The array descriptor for the distributed matrix z. descz[ctxt_ - 1] must 
equal desca[ctxt_ - 1]

work  (local workspace) array of size lwork

lwork  (local )

Size of work array, must be at least 3.
If only eigenvalues are requested:

lwork≥n + max( nb * ( np00 + 1 ), nb * 3 )

If eigenvectors are requested:

lwork≥n + ( np00 + mq00 + nb ) * nb

For definitions of np00 and mq00, see lrwork.
For optimal performance, greater workspace is needed, i.e.
\[ lwork \geq \max( lwork, nhetrd_lwork ) \]
Where \( lwork \) is as defined above, and
\[ nhetrd_lwork = n + 2*( anb+1 )* ( 4*nps+2 ) + ( nps + 1 ) * nps \]
\( ictx = desca[ctxt - 1] \)
\( anb = pjlaenv( ictx, 3, 'PCHETRD', 'L', 0, 0, 0, 0 ) \)
\( sqnpc = \sqrt{ \text{real}( nprow * npcol ) } \)
\( nps = \max( \text{numroc}( n, 1, 0, 0, sqnpc ), 2*anb ) \)
If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \textit{pxerbla}.

\( rwork \)
(local workspace) array of size \( lrwork \)

\( lrwork \)
(local )
Size of \( rwork \), must be at least 3.
See below for definitions of variables used to define \( lrwork \).
If no eigenvectors are requested (\( jobz = 'N' \)) then
\[ lrwork \geq 2 + 5 \ast n + \max( 12 \ast n, nb \ast ( np00 + 1 ) ) \]
If eigenvectors are requested (\( jobz = 'V' \)) then the amount of workspace required is:
\[ lrwork \geq 2 + 5 \ast n + \max( 18 \ast n, np00 \ast mq00 + 2 \ast nb \ast nb ) + (2 + \text{iceil}( neig, nprow*ncol))*n \]

\textbf{NOTE}
\( \text{iceil}(x,y) \) is the ceiling of \( x/y \).

Variable definitions:
\( neig = \) number of eigenvectors requested
\( nb = \) desca[\( mb_ - 1\)] = desca[\( nb_ - 1\)] = descz[\( mb_ - 1\)] = descz[\( nb_ - 1\)]
\( nn = \max( n, nb, 2 ) \)
\( desca[\( rsrc_ - 1\)] = desca[\( csr_ - 1\)] = descz[\( rsrc_ - 1\)] = descz[\( csr_ - 1\)] = 0 \)
\( np00 = \text{numroc}( nn, nb, 0, 0, nprow ) \)
\( mq00 = \text{numroc}( \max( neig, nb, 2 ), nb, 0, 0, npcol ) \)
\( \text{iceil}(x, y) \) is a \textit{ScaLAPACK} function returning \( \text{ceiling}(x/y) \), and \( nprow \) and \( npcol \) can be determined by calling the function \textit{blacs_gridinfo}. 
If \( \text{lrwork} = -1 \), then \( \text{lrwork} \) is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \( \text{pxerbla} \)

\text{liwork} \\
(\text{local workspace}) \text{array of size liwork} \\
\text{liwork} \\
(\text{local}) \quad \text{size of iwork}

Let \( nnp = \max(n, nprow*ncol + 1, 4) \). Then:

\[
\text{liwork} \geq 12 \times nnp + 2 \times n \quad \text{when the eigenvectors are desired}
\]

\[
\text{liwork} \geq 10 \times nnp + 2 \times n \quad \text{when only the eigenvalues have to be computed}
\]

If \( \text{liwork} = -1 \), then \( \text{liwork} \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \)

**OUTPUT Parameters**

\( a \)  
The lower triangle (if \( \text{uplo} = 'L' \)) or the upper triangle (if \( \text{uplo} = 'U' \)) of \( a \), including the diagonal, is destroyed.

\( m \)  
(global) 
Total number of eigenvalues found. \( 0 \leq m \leq n \).

\( nz \)  
(global) 
Total number of eigenvectors computed. \( 0 \leq nz \leq m \).
The number of columns of \( z \) that are filled.
If \( \text{jobz} \neq 'V' \), \( nz \) is not referenced.
If \( \text{jobz} = 'V' \), \( nz = m \)

\( w \)  
(global) \text{array of size } n 
On normal exit, the first \( m \) entries contain the selected eigenvalues in ascending order.

\( z \)  
(\text{local}) \text{array, global size } n \times n \), local size \( \text{lid_z*LOC_c(jz+n-1)} \)
If \( \text{jobz} = 'V' \), then on normal exit the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues.
If \( \text{jobz} = 'N' \), then \( z \) is not referenced.

\( \text{work} \)  
\( \text{work[0]} \) \text{returns workspace adequate workspace to allow optimal performance.}
On return, \( rwork[0] \) contains the optimal amount of workspace required for efficient execution. If \( jobz='N' \) \( rwork[0] \) = optimal amount of workspace required to compute the eigenvalues. If \( jobz='V' \) \( rwork[0] \) = optimal amount of workspace required to compute eigenvalues and eigenvectors.

On return, \( iwork[0] \) contains the amount of integer workspace required.

\( info \) (global )

\( = 0: \) successful exit

\( < 0: \) If the \( i \)-th argument is an array and the \( j \)-th entry had an illegal value, then \( info = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

### Application Notes

The distributed submatrices \( a(ia:*, ja:*) \) and \( z(iz:iz+m-1, jz:jz+n-1) \) must satisfy the following alignment properties:

1. Identical (quadratic) dimension: \( desca[m_-1] = descz[m_-1] = desca[n_-1] = descz[n_-1] \)
3. \( \text{mod}(ia-1, mb_a) = \text{mod}(iz-1, mb_z) = 0 \)

**NOTE**

\( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

### See Also

Overview for details of ScalAPACK array descriptor structures and related notations.

### p?heevx

Computes selected eigenvalues and, optionally, eigenvectors of a Hermitian matrix.

#### Syntax

```c
void pcheevx (char *jobz, char *range, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *vl, float *vu, MKL_INT *il, MKL_INT *iu, float *abstol, MKL_INT *m, MKL_INT *nz, float *w, float *orfac, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *ilwork, MKL_INT *iwork, MKL_INT *iifail, MKL_INT *iclustr, float *gap, MKL_INT *info);
void pzheevx (char *jobz, char *range, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *vl, double *vu, MKL_INT *il, MKL_INT *iu, double *abstol, MKL_INT *m, MKL_INT *nz, double *w, double *orfac, MKL_Complex16 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *ilwork, MKL_INT *iwork, MKL_INT *iifail, MKL_INT *iclustr, double *gap, MKL_INT *info);
```

#### Include Files

- `mkl_scalapack.h`
Description
The p?heevx function computes selected eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix \( A \) by calling the recommended sequence of ScaLAPACK functions. Eigenvalues and eigenvectors can be selected by specifying either a range of values or a range of indices for the desired eigenvalues.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Input Parameters
\( np \) = the number of rows local to a given process.
\( nq \) = the number of columns local to a given process.

\( jobz \) (global) Must be 'N' or 'V'.
Specifies if it is necessary to compute the eigenvectors:
If \( jobz = 'N' \), then only eigenvalues are computed.
If \( jobz = 'V' \), then eigenvalues and eigenvectors are computed.

\( range \) (global) Must be 'A', 'V', or 'I'.
If \( range = 'A' \), all eigenvalues will be found.
If \( range = 'V' \), all eigenvalues in the half-open interval \([vl, vu]\) will be found.
If \( range = 'I' \), the eigenvalues with indices \( il \) through \( iu \) will be found.

\( uplo \) (global) Must be 'U' or 'L'.
Specifies whether the upper or lower triangular part of the Hermitian matrix \( A \) is stored:
If \( uplo = 'U' \), \( a \) stores the upper triangular part of \( A \).
If \( uplo = 'L' \), \( a \) stores the lower triangular part of \( A \).

\( n \) (global) The number of rows and columns of the matrix \( A \) (\( n \geq 0 \)).

\( a \) (local).
Block cyclic array of global size \( n^*n \) and local size \( lld_arcc(ja+n-1) \).
On entry, the Hermitian matrix \( A \).
If \( uplo = 'U' \), only the upper triangular part of \( A \) is used to define the elements of the Hermitian matrix.
If \( uplo = 'L' \), only the lower triangular part of \( A \) is used to define the elements of the Hermitian matrix.
\(ia, ja\) (global) The row and column indices in the global matrix \(A\) indicating the first row and the first column of the submatrix \(A\), respectively.

\(desca\) (global and local) array of size \(dlen_\). The array descriptor for the distributed matrix \(A\). If \(desca[\text{ctxt}_-1]\) is incorrect, \(p?\text{heevx}\) cannot guarantee correct error reporting.

\(vl, vu\) (global) If \(\text{range} = 'V'\), the lower and upper bounds of the interval to be searched for eigenvalues; not referenced if \(\text{range} = 'A'\) or 'I'.

\(il, iu\) (global) If \(\text{range} = 'I'\), the indices of the smallest and largest eigenvalues to be returned.

Constraints:
\[il \geq 1; \min(il, n) \leq iu \leq n.\]

Not referenced if \(\text{range} = 'A'\) or 'V'.

\(abstol\) (global).

If \(\text{jobz}= 'V'\), setting \(abstol\) to \(p?\text{lamch}(context, 'U')\) yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a, b]\) of width less than or equal to \(\text{abstol} + \text{eps} \cdot \max(|a|, |b|)\), where \(\text{eps}\) is the machine precision. If \(\text{abstol}\) is less than or equal to zero, then \(\text{eps} \cdot \text{norm}(T)\) will be used in its place, where \(\text{norm}(T)\) is the 1-norm of the tridiagonal matrix obtained by reducing \(A\) to tridiagonal form.

Eigenvalues are computed most accurately when \(\text{abstol}\) is set to twice the underflow threshold \(2 \cdot p?\text{lamch('S')}\), not zero. If this function returns with \(((\text{mod}(\text{info}, 2) \neq 0)\) or \((\text{mod}(\text{info}/8, 2) \neq 0))\), indicating that some eigenvalues or eigenvectors did not converge, try setting \(abstol\) to \(2 \cdot p?\text{lamch('S')}\).

\textbf{NOTE}
\[\text{mod}(x, y)\] is the integer remainder of \(x/y\).

\(orfac\) (global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within \(tol = orfac \cdot \text{norm}(A)\) of each other are to be reorthogonalized. However, if the workspace is insufficient (see \(lwork\)), \(tol\) may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if \(orfac\) equals zero. A default value of 1.0e-3 is used if \(orfac\) is negative.

\(orfac\) should be identical on all processes.

\(iz, jz\) (global) The row and column indices in the global matrix \(Z\) indicating the first row and the first column of the submatrix \(Z\), respectively.
(global and local) array of size \textit{dlen\_}. The array descriptor for the
distributed matrix \textit{Z}. \textit{descz}[^{\text{ctxt\_ - 1}}] must equal \textit{desca}[^{\text{ctxt\_ - 1}}].

\textit{work}

Array of size \textit{lwork}.

\textit{lwork}

\textit{work} (local). The size of the array \textit{work}.

If only eigenvalues are requested:
\[ lwork \geq n + \max(nb^* (np0 + 1), 3) \]

If eigenvectors are requested:
\[ lwork \geq n + (np0 + mq0 + nb)^* nb \]

with \[ nq0 = \text{numroc}(nn, nb, 0, 0, NPCOL). \]
\[ lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*nb*nb) + \text{ceil}(neig, NPROW*NPCOL)^* nn \]

For optimal performance, greater workspace is needed, that is
\[ lwork \geq \max(lwork, \text{nhetrd\_lwork}) \]

where \textit{lwork} is as defined above, and \textit{nhetrd\_lwork} = \[ n + 2* (\text{anb} +1)^* (4* \text{nps} + 2) + \text{nps} \]

\textit{ictxt} = \text{desca}[^{\text{ctxt\_ - 1}}]

\[ \text{anb} = \text{pjlaenv}(\text{ictxt}, 3, 'pchettrd', 'L', 0, 0, 0, 0) \]

\[ \text{sqnpc} = \sqrt{\text{dble}(\text{NPROW * NPCOL})} \]

\[ \text{nps} = \max(\text{numroc}(n, 1, 0, 0, \text{sqnpc}), 2*\text{anb}) \]

If \textit{lwork} = -1, then \textit{lwork} is global input and a workspace query is
assumed; the function only calculates the size required for optimal
performance for all work arrays. Each of these values is returned in the first
entry of the corresponding work arrays, and no error message is issued by
\text{pxerbla}.

\textit{rwork}

Workspace array of size \textit{lrwork}.

\textit{lrwork}

The size of the array \textit{work}.

See below for definitions of variables used to define \textit{lwork}.

If no eigenvectors are requested (\textit{jobz} = 'N'), then \textit{lrwork} \[ \geq 5*nn + 4*n \.

If eigenvectors are requested (\textit{jobz} = 'V'), then the amount of workspace
required to guarantee that all eigenvectors are computed is:
\[ lrwork \geq 4*n + \max(5*nn, np0*mq0 + 2*nb*nb) + \text{ceil}(neig, NPROW*NPCOL)^* \text{nn} \]

The computed eigenvectors may not be orthogonal if the minimal
workspace is supplied and \textit{orfac} is too small. If you want to guarantee
orthogonality (at the cost of potentially poor performance) you should add
the following values to \textit{lrwork}:
\[ (\text{clustersize}-1)^*n, \]
where \textit{clustersize} is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

\[ |w[k-1],\ldots, w[k+\text{clustersize}-2]| w[j] \leq w[j-1] + \text{orfac} \times 2 \times \text{norm}(A). \]

Variable definitions:

\[ \text{neig} = \text{number of eigenvectors requested}; \]

\[ \text{nb} = \text{desca}[^{mb_1} - 1] = \text{desca}[^{nb_1} - 1] = \text{descz}[^{mb_1} - 1]; \]

\[ \text{nn} = \max(n, \text{NB}, 2); \]

\[ \text{desca}[rsrc_1 - 1] = \text{desca}[nb_1 - 1] = \text{descz}[rsrc_1 - 1] = \]

\[ \text{descz}[csrc_1 - 1] = 0; \]

\[ \text{np} = \text{numroc}(\text{nn}, \text{nb}, 0, 0, \text{NPROW}); \]

\[ \text{mq} = \text{numroc}(\max(\text{neig}, \text{nb}, 2), \text{nb}, 0, 0, \text{NPCOL}); \]

\[ \text{iceil}(x, y) \] is a ScaLAPACK function returning \( \text{ceiling}(x/y) \)

When \textit{lwork} is too small:

If \textit{lwork} is too small to guarantee orthogonality, \texttt{p?heevx} attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If \textit{lwork} is too small to compute all the eigenvectors requested, no computation is performed and \textit{info} = -23 is returned. Note that when \textit{range} = 'V', \texttt{p?heevx} does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when \textit{range} = 'V' and as long as \textit{lwork} is large enough to allow \texttt{p?heevx} to compute the eigenvalues, \texttt{p?heevx} will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality and performance:

If \textit{clustersize} \geq \frac{n}{\sqrt{\text{NPROW} \times \text{NPCOL}}}, then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, \textit{clustersize} = \( n-1 \)) \texttt{p?stein} will perform no better than \texttt{?stein} on 1 processor.

For \textit{clustersize} = \frac{n}{\sqrt{\text{NPROW} \times \text{NPCOL}}} reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For \textit{clustersize} > \frac{n}{\sqrt{\text{NPROW} \times \text{NPCOL}}} execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If \textit{lwork} = -1, then \textit{lwork} is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \texttt{pxerbla}.

\texttt{iwork} \hspace{1cm} (local) Workspace array.

\texttt{liwork} \hspace{1cm} (local), size of \texttt{iwork}.

\[ \text{liwork} \geq 6 \times \text{nnp} \]

Where: \text{nnp} = \max(n, \text{NPROW} \times \text{NPCOL}+1, 4)
If `liwork = -1`, then `liwork` is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

**Output Parameters**

- **a**
  On exit, the lower triangle (if `uplo = 'L'`), or the upper triangle (if `uplo = 'U'`) of `A`, including the diagonal, is overwritten.

- **m**
  (global) The total number of eigenvalues found; `0 ≤ m ≤ n`.

- **nz**
  (global) Total number of eigenvectors computed. `0 ≤ nz ≤ m`.

  The number of columns of `z` that are filled.

  If `jobz ≠ 'V'`, `nz` is not referenced.

  If `jobz = 'V'`, `nz = m` unless the user supplies insufficient space and `p?heevx` is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in `z` (`m×descz[n - 1]`) and sufficient workspace to compute them. (See `lwork`). `p?heevx` is always able to detect insufficient space without computation unless `range = 'V'`.

- **w**
  (global).
  Array of size `n`. The first `m` elements contain the selected eigenvalues in ascending order.

- **z**
  (local).
  Array, global size `n×n`, local size `lld_z×LOCc(jz+n-1)`.

  If `jobz = 'V'`, then on normal exit the first `m` columns of `z` contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of `z` contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.

  If `jobz = 'N'`, then `z` is not referenced.

- **work[0]**
  On exit, returns adequate workspace to allow optimal performance.

- **rwork**
  (local).
  Array of size `lrwork`. On return, `rwork[0]` contains the optimal amount of workspace required for efficient execution.

  If `jobz = 'N'` `rwork[0]` = optimal amount of workspace required to compute eigenvalues efficiently.

  If `jobz = 'V'` `rwork[0]` = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

  If `range = 'V'`, it is assumed that all eigenvectors may be required.

- **iwork[0]**
  (local)
  On return, `iwork[0]` contains the amount of integer workspace required.

- **ifail**
  (global)
Array of size $n$.

If $jobz = 'V'$, then on normal exit, the first $m$ elements of $ifail$ are zero. If $(\text{mod}(\text{info},2)\neq 0)$ on exit, then $ifail$ contains the indices of the eigenvectors that failed to converge.

If $jobz = 'N'$, then $ifail$ is not referenced.

**iclustr**

Array of size $2*NPROW*NPCOL$.

This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see $lwork$, $orfac$ and $info$). Eigenvectors corresponding to clusters of eigenvalues indexed $iclustr[2*i - 2]$ to $iclustr[2*i - 1]$, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

$iclustr$ is a zero terminated array. $(iclustr[2*k - 1] \neq 0$ and $iclustr[2*k] = 0)$ if and only if $k$ is the number of clusters. $iclustr$ is not referenced if $jobz = 'N'$.

**gap**

Array of size $(NPROW*NPCOL)$

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array $iclustr$. As a result, the dot product between eigenvectors corresponding to the $i$-th cluster may be as high as $(C*n)/\text{gap}(i)$ where $C$ is a small constant.

**info**

If $info = 0$, the execution is successful.

If $info < 0$:

If the $i$-th argument is an array and the $j$-entry had an illegal value, then $info = -(i*100+j)$. If the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$:

If $(\text{mod}(\text{info},2)\neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in $ifail$. Ensure $abstol=2.0*p?lamch('U')$

If $(\text{mod}(\text{info}/2,2)\neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array $iclustr$.

If $(\text{mod}(\text{info}/4,2)\neq 0)$, then space limit prevented $p?syevx$ from computing all of the eigenvectors between $vl$ and $vu$. The number of eigenvectors computed is returned in $nz$.

If $(\text{mod}(\text{info}/8,2)\neq 0)$, then $p?stebz$ failed to compute eigenvalues. Ensure $abstol=2.0*p?lamch('U')$.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?gesvd**

Computes the singular value decomposition of a general matrix, optionally computing the left and/or right singular vectors.

**Syntax**

void psgesvd (char *jobu, char *jobvt, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *s, float *u, MKL_INT *iu, MKL_INT *ju, MKL_INT *descu, float *vt, MKL_INT *ivt, MKL_INT *jvt, MKL_INT *descvt, float *work, MKL_INT *lwork, float *rwork, MKL_INT *info);

void pdgesvd (char *jobu, char *jobvt, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *s, double *u, MKL_INT *iu, MKL_INT *ju, MKL_INT *descu, double *vt, MKL_INT *iwt, MKL_INT *jvt, MKL_INT *descvt, double *work, MKL_INT *lwork, double *rwork, MKL_INT *info);

void pcgesvd (char *jobu, char *jobvt, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *s, MKL_Complex8 *u, MKL_INT *iu, MKL_INT *ju, MKL_INT *descu, MKL_Complex8 *vt, MKL_INT *iwt, MKL_INT *jvt, MKL_INT *descvt, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *info);

void pzgesvd (char *jobu, char *jobvt, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *s, MKL_Complex16 *u, MKL_INT *iu, MKL_INT *ju, MKL_INT *descu, MKL_Complex16 *vt, MKL_INT *iwt, MKL_INT *jvt, MKL_INT *descvt, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *info);

**Include Files**

- mkl_scalapack.h

**Description**

The p?gesvd function computes the singular value decomposition (SVD) of an $m$-by-$n$ matrix $A$, optionally computing the left and/or right singular vectors. The SVD is written

$$A = U \Sigma V^T,$$

where $\Sigma$ is an $m$-by-$n$ matrix that is zero except for its $\min(m, n)$ diagonal elements, $U$ is an $m$-by-$m$ orthogonal matrix, and $V$ is an $n$-by-$n$ orthogonal matrix. The diagonal elements of $\Sigma$ are the singular values of $A$ and the columns of $U$ and $V$ are the corresponding right and left singular vectors, respectively. The singular values are returned in array $s$ in decreasing order and only the first $\min(m,n)$ columns of $U$ and rows of $vt = V^T$ are computed.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
NOTE
The distributed submatrix sub(A) must verify certain alignment properties. These expressions must be true:

- \( mb_a = nb_a = nb \)
- \( iroffa = icoffa \)

where:

- \( iroffa = \text{mod}(ia-1, nb) \)
- \( icoffa = \text{mod}(ja-1, nb) \)

Input Parameters

- \( mp = \) number of local rows in \( A \) and \( U \)
- \( nq = \) number of local columns in \( A \) and \( VT \)
- \( size = \) \( \min(m, n) \)
- \( sizeq = \) number of local columns in \( U \)
- \( sizep = \) number of local rows in \( VT \)

- \( jobu \) (global) Specifies options for computing all or part of the matrix \( U \).
  - If \( jobu = 'V' \), the first \( size \) columns of \( U \) (the left singular vectors) are returned in the array \( u \);
  - If \( jobu = 'N' \), no columns of \( U \) (no left singular vectors) are computed.

- \( jobvt \) (global)
  - Specifies options for computing all or part of the matrix \( V^T \).
  - If \( jobvt = 'V' \), the first \( size \) rows of \( V^T \) (the right singular vectors) are returned in the array \( vt \);
  - If \( jobvt = 'N' \), no rows of \( V^T \) (no right singular vectors) are computed.

- \( m \) (global) The number of rows of the matrix \( A \) (\( m \geq 0 \)).
- \( n \) (global) The number of columns in \( A \) (\( n \geq 0 \)).
- \( a \) (local).
  - Block cyclic array, global size \((m, n)\), local size \((mp, nq)\).

- \( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the submatrix \( A \), respectively.

- \( desca \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

- \( iu, ju \) (global) The row and column indices in the global matrix \( U \) indicating the first row and the first column of the submatrix \( U \), respectively.

- \( descu \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( U \).

- \( ivt, jvt \) (global) The row and column indices in the global matrix \( VT \) indicating the first row and the first column of the submatrix \( VT \), respectively.
(global and local) array of size $dlen_\ast$. The array descriptor for the distributed matrix $VT$.

(work) Workspace array of size $lwork$

(lwork) The size of the array $work$;

\[ lwork > 2 + 6\times sizeb + \max(watobd, wbdtosvd), \]

where $sizeb = \max(m, n)$, and $watobd$ and $wbdtosvd$ refer, respectively, to the workspace required to bidiagonalize the matrix $A$ and to go from the bidiagonal matrix to the singular value decomposition $USVT$.

For $watobd$, the following holds:

\[ watobd = \max(\max(wp?lange, wp?gebrd), \max(wp\!\!lared2d, wp\!\!lared1d)), \]


\[ mp = \text{numroc}(m, mb, MYROW, desca[\ast - 1], NPROW), \]
\[ nq = \text{numroc}(n, nb, MYCOL, desca[lld\!\! - 1], NPCOL), \]

the workspaces required for the above subprograms are

\[ wp?lange = mp, \]
\[ wp?lared1d = nq0, \]
\[ wp?lared2d = mp0, \]
\[ wp?gebrd = nb\times(mp + nq + 1) + nq, \]

where $nq0$ and $mp0$ refer, respectively, to the values obtained at $MYCOL = 0$ and $MYROW = 0$. In general, the upper limit for the workspace is given by a workspace required on processor $(0,0)$:

\[ watobd \leq nb\times(mp0 + nq0 + 1) + nq0. \]

In case of a homogeneous process grid this upper limit can be used as an estimate of the minimum workspace for every processor.

For $wbdtosvd$, the following holds:

\[ wbdtosvd = size\times(wantu\times nru + wantvt\times ncvt) + \max(w?bdsqr, \max(wantu\times wp?ormbrqql, wantvt\times wp?ormbrprt)), \]

where

\[ wantu(wantt) = 1, \text{ if left/right singular vectors are wanted, and } \]
\[ wantu(wantt) = 0, \text{ otherwise. } w?bdsqr, wp?ormbrqql, \text{ and } wp?ormbrprt \]
\[ \text{refer respectively to the workspace required for the subprograms } ?bdsqr, \]
\[ p?ormbr(qln), \text{ and } p?ormbr(prt), \text{ where } qln \text{ and } prt \text{ are the values of the } \]
\[ \text{arguments } vect, \text{ side, and } trans \text{ in the call to } p?ormbr. nru \text{ is equal to the } \]
\[ \text{local number of rows of the matrix } U \text{ when distributed 1-dimensional } \]
\[ \text{"column" of processes. Analogously, } ncvt \text{ is equal to the local number of } \]
\[ \text{columns of the matrix } VT \text{ when distributed across 1-dimensional "row" of } \]
\[ \text{processes. Calling the LAPACK procedure } ?bdsqr \text{ requires } \]
\[ w?bdsqr = \max(1, 2\times size + (2\times size - 4)\times \max(wantu, wantvt)) \]
on every processor. Finally,

\[
\begin{align*}
wp\text{ormbrqin} &= \max((nb*(nb-1))/2, (sizeq*mp)*nb)+nb*nb, \\
wp\text{ormbrprt} &= \max((mb*(mb-1))/2, (sizep+nq)*mb)+mb*mb,
\end{align*}
\]

If \text{lwork} = -1, then \text{lwork} is global input and a workspace query is assumed; the function only calculates the minimum size for the work array. The required workspace is returned as the first element of \text{work} and no error message is issued by \text{pxerbla}.

\textit{rwork}

Workspace array of size \(1 + 4*\text{sizeb}\). Not used for \text{p}gesvd and \text{pdgesvd}.

\textbf{Output Parameters}

\textit{a}

On exit, the contents of \textit{a} are destroyed.

\textit{s}

(gridal).

Array of size \textit{size}.

Contains the singular values of \(A\) sorted so that \(s(i) \geq s(i+1)\).

\textit{u}

(local).

local size \(mp*\text{sizeq}\), global size \(m*\text{size}\)

If \textit{jobu} = 'V', \textit{u} contains the first \(\min(m, n)\) columns of \(U\).

If \textit{jobu} = 'N' or 'O', \textit{u} is not referenced.

\textit{vt}

(local).

local size \((\text{sizep, nq})\), global size \((\text{size, n})\)

If \textit{jobvt} = 'V', \textit{vt} contains the first \textit{size} rows of \(V^T\) if \textit{jobu} = 'N', \textit{vt} is not referenced.

\textit{work}

On exit, if \textit{info} = 0, then \textit{work}[0] returns the required minimal size of \textit{lwork}.

\textit{rwork}

On exit, if \textit{info} = 0, then \textit{rwork}[0] returns the required size of \textit{rwork}.

\textit{info}

(global)

If \textit{info} = 0, the execution is successful.

If \textit{info} < 0, if \textit{info} = -i, the \textit{i}th parameter had an illegal value.

If \textit{info} > 0 \(i\), then if \textit{?bdsqr} did not converge,

If \textit{info} = \(\min(m,n) + 1\), then \textit{p}gesvd has detected heterogeneity by finding that eigenvalues were not identical across the process grid. In this case, the accuracy of the results from \textit{p}gesvd cannot be guaranteed.

\textbf{See Also}

\textbf{Overview} for details of ScaLAPACK array descriptor structures and related notations.

\textit{p}\textit{sygvx}

Computes selected eigenvalues and, optionally, eigenvectors of a real generalized symmetric definite eigenproblem.
Syntax

```c
void pssygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , float *vl , float *vu , MKL_INT *il , MKL_INT *iu , float *abstol , MKL_INT *m , MKL_INT *nz , float *w , float *orfac , float *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , float *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *iwork , MKL_INT *ifail , MKL_INT *iwork , MKL_INT *info);

void pdsygvx (MKL_INT *ibtype , char *jobz , char *range , char *uplo , MKL_INT *n ,
double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , double *vl , double *vu , MKL_INT *il , MKL_INT *iu , double *abstol , MKL_INT *m , MKL_INT *nz , double *w , double *orfac , double *z , MKL_INT *iz , MKL_INT *jz , MKL_INT *descz , double *work , MKL_INT *lwork , MKL_INT *iwork , MKL_INT *iwork , MKL_INT *ifail , MKL_INT *iwork , MKL_INT *info);
```

Include Files

- mkl_scalapack.h

Description

The `pssygvx` function computes all the eigenvalues, and optionally, the eigenvectors of a real generalized symmetric-definite eigenproblem, of the form

\[
\text{sub}(A)x = \lambda\text{sub}(B)x , \quad \text{sub}(A)\text{sub}(B)x = \lambda x , \quad \text{or} \quad \text{sub}(B)\text{sub}(A)x = \lambda x .
\]

Here \(x\) denotes eigen vectors, \(\lambda\) (\(\text{lambda}\)) denotes eigenvalues, \(\text{sub}(A)\) denoting \(A(ia:ia+n-1, ja:ja+n-1)\) is assumed to symmetric, and \(\text{sub}(B)\) denoting \(B(ib:ib+n-1, jb:jb+n-1)\) is also positive definite.

Input Parameters

- **ibtype**
  (global) Must be 1 or 2 or 3.
  Specifies the problem type to be solved:
  - If \(ibtype = 1\), the problem type is \(\text{sub}(A)x = \lambda\text{sub}(B)x\);
  - If \(ibtype = 2\), the problem type is \(\text{sub}(A)\text{sub}(B)x = \lambda x\);
  - If \(ibtype = 3\), the problem type is \(\text{sub}(B)\text{sub}(A)x = \lambda x\).

- **jobz**
  (global) Must be 'N' or 'V'.
  - If \(jobz = 'N'\), then compute eigenvalues only.
  - If \(jobz = 'V'\), then compute eigenvalues and eigenvectors.
range
(global) Must be 'A' or 'V' or 'I'.
If range = 'A', the function computes all eigenvalues.
If range = 'V', the function computes eigenvalues in the interval: [vl, vu]
If range = 'I', the function computes eigenvalues with indices il through iu.

uplo
(global) Must be 'U' or 'L'.
If uplo = 'U', arrays a and b store the upper triangles of sub(A) and sub (B);
If uplo = 'L', arrays a and b store the lower triangles of sub(A) and sub (B).

n
(global) The order of the matrices sub(A) and sub (B), n ≥ 0.

a
(local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1). On
entry, this array contains the local pieces of the n-by-n symmetric
distributed matrix sub(A).
If uplo = 'U', the leading n-by-n upper triangular part of sub(A) contains
the upper triangular part of the matrix.
If uplo = 'L', the leading n-by-n lower triangular part of sub(A) contains
the lower triangular part of the matrix.

ia, ja
(global) The row and column indices in the global matrix A indicating the
first row and the first column of the submatrix A, respectively.

desca
(global and local) array of size dlen_. The array descriptor for the
distributed matrix A. If desca[ctxt_ - 1] is incorrect, p?sygvx cannot
guarantee correct error reporting.

b
(local).
Pointer into the local memory to an array of size lld_b*LOCc(jb+n-1). On
entry, this array contains the local pieces of the n-by-n symmetric
distributed matrix sub(B).
If uplo = 'U', the leading n-by-n upper triangular part of sub(B) contains
the upper triangular part of the matrix.
If uplo = 'L', the leading n-by-n lower triangular part of sub(A) contains
the lower triangular part of the matrix.

ib, jb
(global) The row and column indices in the global matrix B indicating the
first row and the first column of the submatrix B, respectively.

descb
(global and local) array of size dlen_. The array descriptor for the
distributed matrix B. descb[ctxt_ - 1] must be equal to desca[ctxt_ - 1].

vl, vu
(global)
If range = 'V', the lower and upper bounds of the interval to be searched
for eigenvalues.
If range = 'A' or 'I', vl and vu are not referenced.

\( il, iu \)  
(global)

If range = 'I', the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: \( il \geq 1, \min(il, n) \leq iu \leq n \)

If range = 'A' or 'V', il and iu are not referenced.

\( abstol \)  
(global)

If jobz='V', setting \( abstol \) to \( p\text{?lamch('S')} \) yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval \([a, b]\) of width less than or equal to

\[
abstol + \text{eps}\cdot\max(|a|,|b|),
\]

where \( \text{eps} \) is the machine precision. If \( abstol \) is less than or equal to zero, then \( \text{eps}\cdot\text{norm}(T) \) will be used in its place, where \( \text{norm}(T) \) is the 1-norm of the tridiagonal matrix obtained by reducing \( A \) to tridiagonal form.

Eigenvalues will be computed most accurately when \( abstol \) is set to twice the underflow threshold \( 2\cdot p\text{?lamch('S')} \) not zero. If this function returns with ((mod(info,2)\neq0) or (mod(info/8,2)\neq0)), indicating that some eigenvalues or eigenvectors did not converge, try setting \( abstol \) to \( 2\cdot p\text{?lamch('S')} \).

**NOTE**

\( \text{mod}(x,y) \) is the integer remainder of \( x/y \).

\( orfac \)  
(global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within \( \text{tol}=\text{orfac}\cdot\text{norm}(A) \) of each other are to be reorthogonalized. However, if the workspace is insufficient (see \( lwork \)), \( \text{tol} \) may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if \( orfac \) equals zero. A default value of 1.0e-3 is used if \( orfac \) is negative. \( orfac \) should be identical on all processes.

\( iz, jz \)  
(global) The row and column indices in the global matrix \( Z \) indicating the first row and the first column of the submatrix \( Z \), respectively.

\( descz \)  
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( Z_\text{descz}[\text{ctxt}_-1] \) must equal \( \text{desca}[\text{ctxt}_-1] \).

\( work \)  
(local)

Workspace array of size \( lwork \)

\( lwork \)  
(local)

Size of the array \( work \). See below for definitions of variables used to define \( lwork \).

If no eigenvectors are requested (jobz = 'N'), then \( lwork \geq 5*n + \max(5*nn, NB*(np0 + 1)) \).
If eigenvectors are requested (jobz = 'V'), then the amount of workspace required to guarantee that all eigenvectors are computed is:

\[ lwork \geq 5*n + \max(5*nn, np0*mq0 + 2*nbp) + \text{ceil}(neig, NPROW*NPCOL)*nn. \]

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and orfac is too small. If you want to guarantee orthogonality at the cost of potentially poor performance you should add the following to lwork:

\[(\text{clustersize}-1)*n,\]

where clustersize is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

\[\{w[k-1], ..., w[k+\text{clustersize}-2]|w[j] \leq w[j-1] + \text{orfac}*2*\text{norm}(A)\}\]

Variable definitions:

\[ neig = \text{number of eigenvectors requested}, \]
\[ nb = \text{desca}[mb - 1] = \text{desca}[nb - 1] = \text{descz}[mb - 1] = \text{descz}[nb - 1], \]
\[ nn = \max(n, nb, 2), \]
\[ \text{desca}[rsrca - 1] = \text{desca}[nb - 1] = \text{descz}[rsrca - 1] = \text{descz}[csrca - 1] = 0, \]
\[ np0 = \text{numroc}(nn, nb, 0, 0, NPROW), \]
\[ mq0 = \text{numroc}(\max(neig, nb, 2), nb, 0, 0, NPCOL) \]

iceil(x, y) is a ScaLAPACK function returning ceiling(x/y)

If lwork is too small to guarantee orthogonality, p?syevx attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If lwork is too small to compute all the eigenvectors requested, no computation is performed and info = -23 is returned.

Note that when range='V', number of requested eigenvectors are not known until the eigenvalues are computed. In this case and if lwork is large enough to compute the eigenvalues, p?sygvx computes the eigenvalues and as many eigenvectors as possible.

Greater performance can be achieved if adequate workspace is provided. In some situations, performance can decrease as the provided workspace increases above the workspace amount shown below:

\[ lwork \geq \max(lwork, 5*n + \text{nSYTRD_LWOPT}, \text{nSYGST_LWOPT}), \]

where lwork, as defined previously, depends upon the number of eigenvectors requested, and

\[ \text{nSYTRD_LWOPT} = n + 2*(anb+1)*(4*nps+2) + (nps+3)*nps \]
\[ \text{nSYGST_LWOPT} = 2*np0*nb + ng0*nb + nb*nb \]
\[ anb = \text{pJLAEVN}(\text{desca}[ctxt - 1], 3, \text{pSYTTRD}', 'L', 0, 0, 0, 0) \]
\[ sqnpc = \text{int}(\sqrt{\text{dble}(NPROW \times NPCOL)}) \]
\[ nps = \max(\text{numroc}(n, 1, 0, 0, sqnpc), 2 \times anb) \]
\[ NB = \text{desca}[mb_- - 1] \]
\[ np0 = \text{numroc}(n, nb, 0, 0, NPROW) \]
\[ nq0 = \text{numroc}(n, nb, 0, 0, NPCOL) \]

\text{numroc} is a ScaLAPACK tool functions;
\text{pjlaenv} is a ScaLAPACK environmental inquiry function

MYROW, MYCOL, NPROW and NPCOL can be determined by calling the function \text{blacs_gridinfo}.

For large \( n \), no extra workspace is needed, however the biggest boost in performance comes for small \( n \), so it is wise to provide the extra workspace (typically less than a Megabyte per process).

If \text{clustersize} \geq n/\sqrt{\text{NPROW} \times \text{NPCOL}} \), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. At the limit (that is, \text{clustersize} = n-1) \text{p?stein} will perform no better than \text{?stein} on a single processor.

For \text{clustersize} = n/\sqrt{\text{NPROW} \times \text{NPCOL}} \) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For \text{clustersize} > n/\sqrt{\text{NPROW} \times \text{NPCOL}} \) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.

If \text{iwork} = -1, then \text{iwork} is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \text{pxerbla}.

\[ \text{iwork} \] (local) Workspace array.

\[ \text{liwork} \] (local), size of \text{iwork}.
\[ \text{liwork} \geq 6 \times nnp \]

Where:
\[ nnp = \max(n, \text{NPROW} \times \text{NPCOL} + 1, 4) \]

If \text{liwork} = -1, then \text{liwork} is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\[ a \]

On exit, 
If \text{jobz} = 'V', and if \text{info} = 0, \text{sub}(A) contains the distributed matrix \( Z \) of eigenvectors. The eigenvectors are normalized as follows:
for \text{ibtype} = 1 or 2, \( Z^T \times \text{sub}(B) \times Z = \text{I} \);
for ibtype = 3, \( Z^T \text{inv}(\text{sub}(B)) Z = i. \)

If jobz = 'N', then on exit the upper triangle (if uplo='U') or the lower triangle (if uplo='L') of sub(A), including the diagonal, is destroyed.

On exit, if info \( \leq n \), the part of sub(B) containing the matrix is overwritten by the triangular factor \( U \) or \( L \) from the Cholesky factorization \( \text{sub}(B) = U^T U \) or \( \text{sub}(B) = L^T L \).

\( b \)

(global) The total number of eigenvalues found, \( 0 \leq m \leq n \).

\( m \)

Total number of eigenvectors computed. \( 0 \leq nz \leq m \). The number of columns of \( z \) that are filled.

If jobz \( \neq 'V' \), nz is not referenced.

If jobz = 'V', nz = m unless the user supplies insufficient space and p?sygvx is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in \( z \) (\( m \leq \text{descz}(n) \)) and sufficient workspace to compute them. (See lwork below.) p?sygvx is always able to detect insufficient space without computation unless range='V'.

\( w \)

(global)

Array of size \( n \). On normal exit, the first \( m \) entries contain the selected eigenvalues in ascending order.

\( z \)

(local).

global size \( n * n \), local size \( \text{lld}\_z * \text{LOCc}(jz+n-1) \).

If jobz = 'V', then on normal exit the first \( m \) columns of \( z \) contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of \( z \) contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in ifail.

If jobz = 'N', then \( z \) is not referenced.

\( \text{work} \)

If jobz='N', work[0] = optimal amount of workspace required to compute eigenvalues efficiently

If jobz = 'V', work[0] = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If range='V', it is assumed that all eigenvectors may be required.

\( \text{ifail} \)

(global)

Array of size \( n \).

\( \text{ifail} \) provides additional information when info\#0

If \( \text{mod}(\text{info}\_16,2) \neq 0 \) then ifail[0] indicates the order of the smallest minor which is not positive definite. If \( \text{mod}(\text{info},2) \neq 0 \) on exit, then ifail contains the indices of the eigenvectors that failed to converge.
If neither of the above error conditions hold and \( \text{jobz} = 'V' \), then the first \( m \) elements of \( \text{ifail} \) are set to zero.

\( \text{iclustr} \) (global)

Array of size \((2*NPROW*NPCOL)\). This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see \( \text{lwork}, \text{orfac} \) and \( \text{info} \)). Eigenvectors corresponding to clusters of eigenvalues indexed \( \text{iclustr}[2*i - 2] \) to \( \text{iclustr}[2*i - 1] \), could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal. \( \text{iclustr} \) is a zero terminated array.

\((\text{iclustr}[2*k - 1]\neq 0. \text{and. iclustr}[2*k]=0) \) if and only if \( k \) is the number of clusters \( \text{iclustr} \) is not referenced if \( \text{jobz} = 'N' \).

\( \text{gap} \) (global)

Array of size \( NPROW*NPCOL \). This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array \( \text{iclustr} \). As a result, the dot product between eigenvectors corresponding to the \( i \)-th cluster may be as high as \( (C*n)/\text{gap}[i - 1] \), where \( C \) is a small constant.

\( \text{info} \) (global)

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} < 0 \): the \( i \)-th argument is an array and the \( j \)-entry had an illegal value, then \( \text{info} = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

If \( \text{info} > 0 \):

If \( (\text{mod(}\text{info},2)\neq 0) \), then one or more eigenvectors failed to converge. Their indices are stored in \( \text{ifail} \).

If \( (\text{mod(}\text{info},2,2)\neq 0) \), then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array \( \text{iclustr} \).

If \( (\text{mod(}\text{info},4,2)\neq 0) \), then space limit prevented \( \text{p?sygvx} \) from computing all of the eigenvectors between \( \text{vl} \) and \( \text{vu} \). The number of eigenvectors computed is returned in \( \text{nz} \).

If \( (\text{mod(}\text{info},8,2)\neq 0) \), then \( \text{p?stebz} \) failed to compute eigenvalues.

If \( (\text{mod(}\text{info},16,2)\neq 0) \), then \( B \) was not positive definite. \( \text{ifail}(1) \) indicates the order of the smallest minor which is not positive definite.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\( \text{p?hegvx} \)

Computes selected eigenvalues and, optionally, eigenvectors of a complex generalized Hermitian positive-definite eigenproblem.
Syntax

void pchegvx (MKL_INT *ibtype, char *jobz, char *range, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *descA, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descB, float *vl, float *vu, MKL_INT *il, MKL_INT *iu, float *abstol, MKL_INT *m, MKL_INT *nz, float *w, float *orfac, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descZ, float *vl, float *vu, MKL_INT *il, MKL_INT *iu, float *abstol, MKL_INT *m, MKL_INT *nz, float *w, float *orfac, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descZ, MKL_Complex8 *work, MKL_INT *lwork, float *rwork, MKL_INT *lrwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *ifail, MKL_INT *iclustr, float *gap, MKL_INT *info);

void pzhegvx (MKL_INT *ibtype, char *jobz, char *range, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *descA, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descB, double *vl, double *vu, MKL_INT *il, MKL_INT *iu, double *abstol, MKL_INT *m, MKL_INT *nz, double *w, double *orfac, MKL_Complex16 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descZ, MKL_Complex16 *work, MKL_INT *lwork, double *rwork, MKL_INT *lrwork, MKL_INT *iwork, MKL_INT *liwork, MKL_INT *ifail, MKL_INT *iclustr, double *gap, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?hegvx function computes all the eigenvalues, and optionally, the eigenvectors of a complex generalized Hermitian positive-definite eigenproblem, of the form

\[ \text{sub}(A) \ast x = \lambda \ast \text{sub}(B) \ast x, \text{ or } \text{sub}(B) \ast \text{sub}(A) \ast x = \lambda \ast x. \]

Here \( \text{sub}(A) \) denoting \( A(ia:ia+n-1, ja:ja+n-1) \) and \( \text{sub}(B) \) are assumed to be Hermitian and \( \text{sub}(B) \) denoting \( B(ib:ib+n-1, jb:jb+n-1) \) is also positive definite.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

ibtype

(global) Must be 1 or 2 or 3.

Specifies the problem type to be solved:
If \( ibtype = 1 \), the problem type is
\[ \text{sub}(A) \ast x = \lambda \ast \text{sub}(B) \ast x; \]
If \( ibtype = 2 \), the problem type is
\[ \text{sub}(B) \ast \text{sub}(A) \ast x = \lambda \ast x; \]
If \( ibtype = 3 \), the problem type is
sub(B)*sub(A)*x = lambda*x.

jobz (global) Must be 'N' or 'V'.
If jobz = 'N', then compute eigenvalues only.
If jobz = 'V', then compute eigenvalues and eigenvectors.

range (global) Must be 'A' or 'V' or 'I'.
If range = 'A', the function computes all eigenvalues.
If range = 'V', the function computes eigenvalues in the interval: [vl, vu]
If range = 'I', the function computes eigenvalues with indices il through iu.

uplo (global) Must be 'U' or 'L'.
If uplo = 'U', arrays a and b store the upper triangles of sub(A) and sub(B);
If uplo = 'L', arrays a and b store the lower triangles of sub(A) and sub(B).

n (global)
The order of the matrices sub(A) and sub(B) (n ≥ 0).

a (local)
Pointer into the local memory to an array of size lld_a*LOCc(ja+n-1). On entry, this array contains the local pieces of the n-by-n Hermitian distributed matrix sub(A). If uplo = 'U', the leading n-by-n upper triangular part of sub(A) contains the upper triangular part of the matrix. If uplo = 'L', the leading n-by-n lower triangular part of sub(A) contains the lower triangular part of the matrix.

ia, ja (global)
The row and column indices in the global matrix A indicating the first row and the first column of the submatrix A, respectively.

desca (global and local) array of size dlen_.
The array descriptor for the distributed matrix A. If desca[ctxt_ - 1] is incorrect, p?hegvx cannot guarantee correct error reporting.

b (local).
Pointer into the local memory to an array of size lld_b*LOCc(jb+n-1). On entry, this array contains the local pieces of the n-by-n Hermitian distributed matrix sub(B).
If uplo = 'U', the leading n-by-n upper triangular part of sub(B) contains the upper triangular part of the matrix.
If uplo = 'L', the leading n-by-n lower triangular part of sub(B) contains the lower triangular part of the matrix.

ib, jb (global)
The row and column indices in the global matrix $B$ indicating the first row and the first column of the submatrix $B$, respectively.

descb

(global and local) array of size $dlen_\cdot$

The array descriptor for the distributed matrix $B$. $descb[ctxt_\cdot - 1]$ must be equal to $desca[ctxt_\cdot - 1]$.

$v\ell, vu$

(global)

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues.

If $range = 'A'$ or $'I'$, $v\ell$ and $vu$ are not referenced.

$il, iu$

(global)

If $range = 'I'$, the indices in ascending order of the smallest and largest eigenvalues to be returned. Constraint: $il \geq 1$, $\min(il, n) \leq iu \leq n$

If $range = 'A'$ or $'V'$, $il$ and $iu$ are not referenced.

$abstol$

(global)

If $jobz='V'$, setting $abstol$ to $\max\{2*p?lamch('S')\}$ yields the most orthogonal eigenvectors.

The absolute error tolerance for the eigenvalues. An approximate eigenvalue is accepted as converged when it is determined to lie in an interval $[a, b]$ of width less than or equal to

$$abstol + eps*\max(|a|,|b|),$$

where $eps$ is the machine precision. If $abstol$ is less than or equal to zero, then $eps*\max(T)$ will be used in its place, where $\max(T)$ is the 1-norm of the tridiagonal matrix obtained by reducing $A$ to tridiagonal form.

Eigenvalues will be computed most accurately when $abstol$ is set to twice the underflow threshold $2*p?lamch('S')$ not zero. If this function returns with $(\mod(info,2)\neq0)$.or. $(\mod(info,8,2)\neq0))$, indicating that some eigenvalues or eigenvectors did not converge, try setting $abstol$ to $2*p?lamch('S')$.

**NOTE**

$\mod(x,y)$ is the integer remainder of $x/y$.

$orfac$

(global).

Specifies which eigenvectors should be reorthogonalized. Eigenvectors that correspond to eigenvalues which are within $tol=orfac*\max(A)$ of each other are to be reorthogonalized. However, if the workspace is insufficient (see $lwork$), $tol$ may be decreased until all eigenvectors to be reorthogonalized can be stored in one process. No reorthogonalization will be done if $orfac$ equals zero. A default value of 1.0E-3 is used if $orfac$ is negative. $orfac$ should be identical on all processes.

$iz, jz$

(global) The row and column indices in the global matrix $Z$ indicating the first row and the first column of the submatrix $Z$, respectively.
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( Z \). The array descriptor for the distributed matrix \( Z \) must equal \( desca[ctxt_-1] \).

\textbf{work}

(local)

Workspace array of size \( lwork \)

\textbf{lwork}

(local).

The size of the array \( work \).

If only eigenvalues are requested:

\[ lwork \geq n \times \max(NB \times (np0 + 1), 3) \]

If eigenvectors are requested:

\[ lwork \geq n + (np0 + mq0 + NB) \times NB \]

with \( np0 = \text{numroc}(nn, NB, 0, 0, NPCOL) \).

For optimal performance, greater workspace is needed, that is

\[ lwork \geq \max(lwork, n, \text{nhetrd_lwopt}, \text{nhegst_lwopt}) \]

where \( lwork \) is as defined above, and

\[ \text{nhetrd_lwopt} = 2 \times (\text{anb} + 1) \times (4 \times \text{nps} + 2) \times (\text{nps} + 1) \times \text{nps}; \]

\[ \text{nhegst_lwopt} = 2 \times \text{np0} \times \text{nb} + \text{np0} \times \text{nb} + \text{nb} \times \text{nb} \]

\[ \text{nb} = \text{desca}[mb_-1] \]

\[ np0 = \text{numroc}(n, nb, 0, 0, NPROW) \]

\[ npq0 = \text{numroc}(n, nb, 0, 0, NPCOL) \]

\[ \text{ictxt} = \text{desca}[ctxt_-1] \]

\[ \text{anb} = \text{pjlaenv}(\text{ictxt}, 3, 'p?hettrd', 'L', 0, 0, 0, 0) \]

\[ \text{sqnpc} = \sqrt{\text{dble}(NPROW \times NPCOL)} \]

\[ \text{nps} = \max(\text{numroc}(n, 1, 0, 0, \text{sqnpc}), 2 \times \text{anb}) \]

\textbf{numroc} is a ScaLAPACK tool functions;

\textbf{pjlaenv} is a ScaLAPACK environmental inquiry function \textbf{MYROW}, \textbf{MYCOL}, \textbf{NPROW} and \textbf{NPCOL} can be determined by calling the function \textbf{blacs_gridinfo}.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \textbf{pxerbla}.

\textbf{rwork}

(local)

Workspace array of size \( lrwork \).

\textbf{lrwork}

(local) The size of the array \( rwork \).

See below for definitions of variables used to define \( lrwork \).

If no eigenvectors are requested (\( jobz = 'N' \)), then \( lrwork \geq 5 \times nn + 4 \times n \)
If eigenvectors are requested (\texttt{jobz = 'V'}), then the amount of workspace required to guarantee that all eigenvectors are computed is:

\[ lrwork \geq 4n + \max(5nn, np0*mq0) + \text{iceil}(\text{neig}, \text{NPROW}*\text{NPCOL})*nn \]

The computed eigenvectors may not be orthogonal if the minimal workspace is supplied and \texttt{orfac} is too small. If you want to guarantee orthogonality (at the cost of potentially poor performance) you should add the following value to \texttt{lrwork}:

\[ (\text{clustersize}-1)*n, \]

where \texttt{clustersize} is the number of eigenvalues in the largest cluster, where a cluster is defined as a set of close eigenvalues:

\[ \{w[k - 1], ..., w[k+\text{clustersize} - 2]|w[j] \leq w[j-1]+\text{orfac}*2*\text{norm}(A)\} \]

Variable definitions:

\texttt{neig} = number of eigenvectors requested;

\texttt{nb} = \texttt{desca}[mb - 1] = \texttt{desca}[nb - 1] = \texttt{descz}[mb - 1] = \texttt{descz}[nb - 1];

\texttt{nn} = \max(n, nb, 2);


\texttt{np0 = numroc(nn, nb, 0, 0, NPROW)};

\texttt{mq0 = numroc(max(neig, nb, 2), nb, 0, 0, NPCOL)};

\texttt{iceil(x, y)} is a ScaLAPACK function returning ceiling(x/y).

When \texttt{lrwork} is too small:

If \texttt{lwork} is too small to guarantee orthogonality, \texttt{p?hegvx} attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues.

If \texttt{lwork} is too small to compute all the eigenvectors requested, no computation is performed and \texttt{info} = -25 is returned. Note that when \texttt{range='V'}, \texttt{p?hegvx} does not know how many eigenvectors are requested until the eigenvalues are computed. Therefore, when \texttt{range='V'} and as long as \texttt{lwork} is large enough to allow \texttt{p?hegvx} to compute the eigenvalues, \texttt{p?hegvx} will compute the eigenvalues and as many eigenvectors as it can.

Relationship between workspace, orthogonality & performance:

If \texttt{clustersize} > \(n/\sqrt{\text{NPROW}*\text{NPCOL}}\), then providing enough space to compute all the eigenvectors orthogonally will cause serious degradation in performance. In the limit (that is, \texttt{clustersize} = \(n-1\)) \texttt{p?stein} will perform no better than \texttt{?stein} on 1 processor.

For \texttt{clustersize} = \(n/\sqrt{\text{NPROW}*\text{NPCOL}}\) reorthogonalizing all eigenvectors will increase the total execution time by a factor of 2 or more.

For \texttt{clustersize}>\(n/\sqrt{\text{NPROW}*\text{NPCOL}}\) execution time will grow as the square of the cluster size, all other factors remaining equal and assuming enough workspace. Less workspace means less reorthogonalization but faster execution.
If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the size required for optimal performance for all work arrays. Each of these values is returned in the first entry of the corresponding work arrays, and no error message is issued by \( \text{p} \text{xerbla} \).

\( iwork \) (local) Workspace array.

\( liwork \), size of \( iwork \).

\( liwork \geq 6 \times nnp \)

Where: \( nnp = \max(n, \text{NPROW} \times \text{NPCOL} + 1, 4) \)

If \( liwork = -1 \), then \( liwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{p} \text{xerbla} \).

Output Parameters

\( a \) On exit, if \( \text{jobz} = \text{'}V\text{'} \), then if \( \text{info} = 0 \), sub(\( A \)) contains the distributed matrix \( Z \) of eigenvectors.

The eigenvectors are normalized as follows:

If \( ibtype = 1 \) or 2, then \( Z^* \text{sub}(B) Z = I \);

If \( ibtype = 3 \), then \( Z^* \text{inv(sub}(B)) Z = I \).

If \( \text{jobz} = \text{'}N\text{'} \), then on exit the upper triangle (if \( \text{uplo} = \text{'}U\text{'}) or the lower triangle (if \( \text{uplo} = \text{'}L\text{'}) of sub(\( A \)), including the diagonal, is destroyed.

\( b \) On exit, if \( \text{info} \leq n \), the part of sub(\( B \)) containing the matrix is overwritten by the triangular factor \( U \) or \( L \) from the Cholesky factorization sub(\( B \)) = \( U^* U \), or sub(\( B \)) = \( L L^* \).

\( m \) (global) The total number of eigenvalues found, \( 0 \leq m \leq n \).

\( nz \) (global) Total number of eigenvectors computed. \( 0 < nz < m \). The number of columns of \( z \) that are filled.

If \( \text{jobz} \neq \text{'}V\text{'} \), \( nz \) is not referenced.

If \( \text{jobz} = \text{'}V\text{'} \), \( nz = m \) unless the user supplies insufficient space and \( \text{p} \text{hegvx} \) is not able to detect this before beginning computation. To get all the eigenvectors requested, the user must supply both sufficient space to hold the eigenvectors in \( z \) (\( m \leq \text{descz}[n_--1] \)) and sufficient workspace to compute them. (See \( lwork \) below.) The function \( \text{p} \text{hegvx} \) is always able to detect insufficient space without computation unless \( \text{range} = \text{'}V\text{'} \).

\( w \) (global)

Array of size \( n \). On normal exit, the first \( m \) entries contain the selected eigenvalues in ascending order.

\( z \) (local).

Global size \( n \times n \), local size \( \text{ld}_z \times \text{LOCc}(jz+n-1) \).
If `jobz = 'V'`, then on normal exit the first `m` columns of `z` contain the orthonormal eigenvectors of the matrix corresponding to the selected eigenvalues. If an eigenvector fails to converge, then that column of `z` contains the latest approximation to the eigenvector, and the index of the eigenvector is returned in `ifail`.

If `jobz = 'N'`, then `z` is not referenced.

**work**
On exit, `work[0]` returns the optimal amount of workspace.

**rwork**
On exit, `rwork[0]` contains the amount of workspace required for optimal efficiency

- If `jobz='N' rwork[0] = optimal amount of workspace required to compute eigenvalues efficiently
- If `jobz='V' rwork[0] = optimal amount of workspace required to compute eigenvalues and eigenvectors efficiently with no guarantee on orthogonality.

If `range='V'`, it is assumed that all eigenvectors may be required when computing optimal workspace.

**ifail**
Array of size `n`.

- `ifail` provides additional information when `info≠0`
- If `(mod(info,16,2)≠0), then ifail[0] indicates the order of the smallest minor which is not positive definite.
- If `(mod(info,2)≠0) on exit, then ifail[0] contains the indices of the eigenvectors that failed to converge.
- If neither of the above error conditions are held, and `jobz = 'V'`, then the first `m` elements of `ifail` are set to zero.

**iclustr**
Array of size `(2*NPROW*NPCOL)`. This array contains indices of eigenvectors corresponding to a cluster of eigenvalues that could not be reorthogonalized due to insufficient workspace (see `lwork`, `orfac` and `info`). Eigenvectors corresponding to clusters of eigenvalues indexed `iclustr(2*i-1)` to `iclustr(2*i)`, could not be reorthogonalized due to lack of workspace. Hence the eigenvectors corresponding to these clusters may not be orthogonal.

- `iclustr()` is a zero terminated array. `(iclustr(2*k) ≠0 and clustr(2*k+1)=0) if and only if `k` is the number of clusters.
- `iclustr` is not referenced if `jobz = 'N'`.

**gap**
Array of size `NPROW*NPCOL`.

This array contains the gap between eigenvalues whose eigenvectors could not be reorthogonalized. The output values in this array correspond to the clusters indicated by the array `iclustr`. As a result, the dot product between eigenvectors corresponding to the `i`-th cluster may be as high as \((C*n)/gap(i)\), where `C` is a small constant.
If $info = 0$, the execution is successful.

If $info < 0$: the $i$-th argument is an array and the $j$-entry had an illegal value, then $info = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

If $info > 0$:

If $(\text{mod}(info, 2) \neq 0)$, then one or more eigenvectors failed to converge. Their indices are stored in $ifail$.

If $(\text{mod}(info, 2, 2) \neq 0)$, then eigenvectors corresponding to one or more clusters of eigenvalues could not be reorthogonalized because of insufficient workspace. The indices of the clusters are stored in the array $iclustr$.

If $(\text{mod}(info/4, 2) \neq 0)$, then space limit prevented $p$sygvx from computing all of the eigenvectors between $vl$ and $vu$. The number of eigenvectors computed is returned in $nz$.

If $(\text{mod}(info/8, 2) \neq 0)$, then $p$stebz failed to compute eigenvalues.

If $(\text{mod}(info/16, 2) \neq 0)$, then $B$ was not positive definite. $ifail(1)$ indicates the order of the smallest minor which is not positive definite.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

### ScaLAPACK Auxiliary Routines

#### ScaLAPACK Auxiliary Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p$lacgv</td>
<td>$c, z$</td>
<td>Conjugates a complex vector.</td>
</tr>
<tr>
<td>$p$max1</td>
<td>$c, z$</td>
<td>Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS $p$amax, but using the absolute value to the real part).</td>
</tr>
<tr>
<td>pmpcol</td>
<td>$s, d$</td>
<td>Finds the collaborators of a process.</td>
</tr>
<tr>
<td>pmpim2</td>
<td>$s, d$</td>
<td>Computes the eigenpair range assignments for all processes.</td>
</tr>
<tr>
<td>$p$combamax1</td>
<td>$c, z$</td>
<td>Finds the element with maximum real part absolute value and its corresponding global index.</td>
</tr>
<tr>
<td>$p$sum1</td>
<td>$s c, dz$</td>
<td>Forms the 1-norm of a complex vector similar to Level 1 PBLAS $p$asum, but using the true absolute value.</td>
</tr>
<tr>
<td>$p$dbtrsv</td>
<td>$s, d, c, z$</td>
<td>Computes an $LU$ factorization of a general tridiagonal matrix with no pivoting. The routine is called by $p$dbtrs.</td>
</tr>
<tr>
<td>$p$dttrsv</td>
<td>$s, d, c, z$</td>
<td>Computes an $LU$ factorization of a general band matrix, using partial pivoting with row interchanges. The routine is called by $p$dttrs.</td>
</tr>
<tr>
<td>$p$gebal</td>
<td>$s, d$</td>
<td>Balances a general real/complex matrix.</td>
</tr>
<tr>
<td>Routine Name</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>p?gebd2</td>
<td>s, d, c, z</td>
<td>Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).</td>
</tr>
<tr>
<td>p?gehd2</td>
<td>s, d, c, z</td>
<td>Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).</td>
</tr>
<tr>
<td>p?gelq2</td>
<td>s, d, c, z</td>
<td>Computes an $LQ$ factorization of a general rectangular matrix (unblocked algorithm).</td>
</tr>
<tr>
<td>p?geql2</td>
<td>s, d, c, z</td>
<td>Computes a $QL$ factorization of a general rectangular matrix (unblocked algorithm).</td>
</tr>
<tr>
<td>p?geqr2</td>
<td>s, d, c, z</td>
<td>Computes a $QR$ factorization of a general rectangular matrix (unblocked algorithm).</td>
</tr>
<tr>
<td>p?gerq2</td>
<td>s, d, c, z</td>
<td>Computes an $RQ$ factorization of a general rectangular matrix (unblocked algorithm).</td>
</tr>
<tr>
<td>p?getf2</td>
<td>s, d, c, z</td>
<td>Computes an $LU$ factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).</td>
</tr>
<tr>
<td>p?labrd</td>
<td>s, d, c, z</td>
<td>Reduces the first $nb$ rows and columns of a general rectangular matrix $A$ to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of $A$.</td>
</tr>
<tr>
<td>p?lacon</td>
<td>s, d, c, z</td>
<td>Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.</td>
</tr>
<tr>
<td>p?laconsb</td>
<td>s, d</td>
<td>Looks for two consecutive small subdiagonal elements.</td>
</tr>
<tr>
<td>p?lacp2</td>
<td>s, d, c, z</td>
<td>Copies all or part of a distributed matrix to another distributed matrix.</td>
</tr>
<tr>
<td>p?lacp3</td>
<td>s, d</td>
<td>Copies from a global parallel array into a local replicated array or vice versa.</td>
</tr>
<tr>
<td>p?lacpy</td>
<td>s, d, c, z</td>
<td>Copies all or part of one two-dimensional array to another.</td>
</tr>
<tr>
<td>p?laevswp</td>
<td>s, d, c, z</td>
<td>Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.</td>
</tr>
<tr>
<td>p?lahrd</td>
<td>s, d, c, z</td>
<td>Reduces the first $nb$ columns of a general rectangular matrix $A$ so that elements below the $k^{th}$ subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of $A$.</td>
</tr>
<tr>
<td>p?laiect</td>
<td>s, d, c, z</td>
<td>Exploits IEEE arithmetic to accelerate the computations of eigenvalues.</td>
</tr>
<tr>
<td>p?lamve</td>
<td>s, d</td>
<td>Copies all or part of one two-dimensional distributed array to another.</td>
</tr>
<tr>
<td>p?lange</td>
<td>s, d, c, z</td>
<td>Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.</td>
</tr>
<tr>
<td>p?lanhs</td>
<td>s, d, c, z</td>
<td>Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.</td>
</tr>
<tr>
<td>Routine Name</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>p?lansy, p?lanhe</td>
<td>s, d, c, z/c, z</td>
<td>Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element of a real symmetric or complex Hermitian matrix.</td>
</tr>
<tr>
<td>p?lantr</td>
<td>s, d, c, z</td>
<td>Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.</td>
</tr>
<tr>
<td>p?lapiv</td>
<td>s, d, c, z</td>
<td>Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.</td>
</tr>
<tr>
<td>p?laqge</td>
<td>s, d, c, z</td>
<td>Scales a general rectangular matrix, using row and column scaling factors computed by p?geequ.</td>
</tr>
<tr>
<td>p?laqr0</td>
<td>s, d</td>
<td>Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.</td>
</tr>
<tr>
<td>p?laqr1</td>
<td>s, d</td>
<td>Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.</td>
</tr>
<tr>
<td>p?laqr2</td>
<td>s, d</td>
<td>Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).</td>
</tr>
<tr>
<td>p?laqr3</td>
<td>s, d</td>
<td>Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).</td>
</tr>
<tr>
<td>p?laqr5</td>
<td>s, d</td>
<td>Performs a single small-bulge multi-shift QR sweep.</td>
</tr>
<tr>
<td>p?laqsy</td>
<td>s, d, c, z</td>
<td>Scales a symmetric/Hermitian matrix, using scaling factors computed by p?poequ.</td>
</tr>
<tr>
<td>p?lared1d</td>
<td>s, d</td>
<td>Redistributes an array assuming that the input array bycol is distributed across rows and that all process columns contain the same copy of bycol.</td>
</tr>
<tr>
<td>p?lared2d</td>
<td>s, d</td>
<td>Redistributes an array assuming that the input array byrow is distributed across columns and that all process rows contain the same copy of byrow.</td>
</tr>
<tr>
<td>p?larf</td>
<td>s, d, c, z</td>
<td>Applies an elementary reflector to a general rectangular matrix.</td>
</tr>
<tr>
<td>p?larfb</td>
<td>s, d, c, z</td>
<td>Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.</td>
</tr>
<tr>
<td>p?larfc</td>
<td>c, z</td>
<td>Applies the conjugate transpose of an elementary reflector to a general matrix.</td>
</tr>
<tr>
<td>p?larfg</td>
<td>s, d, c, z</td>
<td>Generates an elementary reflector (Householder matrix).</td>
</tr>
<tr>
<td>p?larft</td>
<td>s, d, c, z</td>
<td>Forms the triangular vector T of a block reflector $H=I-VTV^T$.</td>
</tr>
<tr>
<td>p?larz</td>
<td>s, d, c, z</td>
<td>Applies an elementary reflector as returned by p?tzrzf to a general matrix.</td>
</tr>
<tr>
<td>p?larzb</td>
<td>s, d, c, z</td>
<td>Applies a block reflector or its transpose/conjugate-transpose as returned by p?tzrzf to a general matrix.</td>
</tr>
<tr>
<td>Routine Name</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>p?larzc</td>
<td>c, z</td>
<td>Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by p?tzrzf to a general matrix.</td>
</tr>
<tr>
<td>p?larzt</td>
<td>s, d, c, z</td>
<td>Forms the triangular factor $T$ of a block reflector $H=I-VTV^H$ as returned by p?tzrzf.</td>
</tr>
<tr>
<td>p?lascl</td>
<td>s, d, c, z</td>
<td>Multiplies a general rectangular matrix by a real scalar defined as $C_{to}/C_{from}$.</td>
</tr>
<tr>
<td>p?laset</td>
<td>s, d, c, z</td>
<td>Initializes the off-diagonal elements of a matrix to $\alpha$ and the diagonal elements to $\beta$.</td>
</tr>
<tr>
<td>p?lasmsub</td>
<td>s, d</td>
<td>Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.</td>
</tr>
<tr>
<td>p?lassq</td>
<td>s, d, c, z</td>
<td>Updates a sum of squares represented in scaled form.</td>
</tr>
<tr>
<td>p?laswp</td>
<td>s, d, c, z</td>
<td>Performs a series of row interchanges on a general rectangular matrix.</td>
</tr>
<tr>
<td>p?latra</td>
<td>s, d, c, z</td>
<td>Computes the trace of a general square distributed matrix.</td>
</tr>
<tr>
<td>p?latrd</td>
<td>s, d, c, z</td>
<td>Reduces the first $nb$ rows and columns of a symmetric/Hermitian matrix $A$ to real tridiagonal form by an orthogonal/unitary similarity transformation.</td>
</tr>
<tr>
<td>p?latrz</td>
<td>s, d, c, z</td>
<td>Reduces an upper trapezoidal matrix to upper triangular form by means of orthogonal/unitary transformations.</td>
</tr>
<tr>
<td>p?lauu2</td>
<td>s, d, c, z</td>
<td>Computes the product $UU^H$ or $L^H L$, where $U$ and $L$ are upper or lower triangular matrices (local unblocked algorithm).</td>
</tr>
<tr>
<td>p?lauum</td>
<td>s, d, c, z</td>
<td>Computes the product $UU^H$ or $L^H L$, where $U$ and $L$ are upper or lower triangular matrices.</td>
</tr>
<tr>
<td>p?lawil</td>
<td>s, d</td>
<td>Forms the Wilkinson transform.</td>
</tr>
<tr>
<td>p?org2l/p?ung2l</td>
<td>s, d, c, z</td>
<td>Generates all or part of the orthogonal/unitary matrix $Q$ from a QL factorization determined by p?geqlf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?org2r/p?ung2r</td>
<td>s, d, c, z</td>
<td>Generates all or part of the orthogonal/unitary matrix $Q$ from a QR factorization determined by p?geqrf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?orgl2/p?ungl2</td>
<td>s, d, c, z</td>
<td>Generates all or part of the orthogonal/unitary matrix $Q$ from an LQ factorization determined by p?gelqf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?orgr2/p?ungr2</td>
<td>s, d, c, z</td>
<td>Generates all or part of the orthogonal/unitary matrix $Q$ from an RQ factorization determined by p?gerqf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?orm2l/p?unm2l</td>
<td>s, d, c, z</td>
<td>Multiplies a general matrix by the orthogonal/unitary matrix from a QL factorization determined by p?geqlf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?orm2r/p?unm2r</td>
<td>s, d, c, z</td>
<td>Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by p?geqrf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?orml2/p?unml2</td>
<td>s, d, c, z</td>
<td>Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).</td>
</tr>
<tr>
<td>p?ormr2/p?unmr2</td>
<td>s, d, c, z</td>
<td>Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by p?gerqf (unblocked algorithm).</td>
</tr>
<tr>
<td>Routine Name</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>p?pbtrsv</td>
<td>s, d, c, z</td>
<td>Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by p?pbtrf.</td>
</tr>
<tr>
<td>p?pttrsv</td>
<td>s, d, c, z</td>
<td>Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by p?pttrf.</td>
</tr>
<tr>
<td>p?potf2</td>
<td>s, d, c, z</td>
<td>Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).</td>
</tr>
<tr>
<td>p?rot</td>
<td>s, d</td>
<td>Applies a planar rotation to two distributed vectors.</td>
</tr>
<tr>
<td>p?rsc1</td>
<td>s, d, cs, zd</td>
<td>Multiplies a vector by the reciprocal of a real scalar.</td>
</tr>
<tr>
<td>p?sygs2/p?hegs2</td>
<td>s, d, c, z</td>
<td>Reduces a symmetric/Hermitian positive-definite generalized eigenproblem to standard form, using the factorization results obtained from p?potrf (local unblocked algorithm).</td>
</tr>
<tr>
<td>p?sytd2/p?hetd2</td>
<td>s, d, c, z</td>
<td>Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).</td>
</tr>
<tr>
<td>p?trord</td>
<td>s, d</td>
<td>Reorders the Schur factorization of a general matrix.</td>
</tr>
<tr>
<td>p?trsen</td>
<td>s, d</td>
<td>Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.</td>
</tr>
<tr>
<td>p?trti2</td>
<td>s, d, c, z</td>
<td>Computes the inverse of a triangular matrix (local unblocked algorithm).</td>
</tr>
<tr>
<td>?lamsh</td>
<td>s, d</td>
<td>Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.</td>
</tr>
<tr>
<td>?laqr6</td>
<td>s, d</td>
<td>Performs a single small-bulge multi-shift QR sweep collecting the transformations.</td>
</tr>
<tr>
<td>?larl1va</td>
<td>s, d</td>
<td>Computes scaled eigenvector corresponding to given eigenvalue.</td>
</tr>
<tr>
<td>?laref</td>
<td>s, d</td>
<td>Applies Householder reflectors to matrices on either their rows or columns.</td>
</tr>
<tr>
<td>?larrb2</td>
<td>s, d</td>
<td>Provides limited bisection to locate eigenvalues for more accuracy.</td>
</tr>
<tr>
<td>?larrd2</td>
<td>s, d</td>
<td>Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.</td>
</tr>
<tr>
<td>?larre2</td>
<td>s, d</td>
<td>Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.</td>
</tr>
<tr>
<td>?larre2a</td>
<td>s, d</td>
<td>Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.</td>
</tr>
<tr>
<td>?larrf2</td>
<td>s, d</td>
<td>Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.</td>
</tr>
<tr>
<td>?larrv2</td>
<td>s, d</td>
<td>Computes the eigenvectors of the tridiagonal matrix $T = L^*D^*L^T$ given $L$, $D$ and the eigenvalues of $L^*D^*L^T$.</td>
</tr>
<tr>
<td>Routine Name</td>
<td>Data Types</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>?lasorte</td>
<td>s, d</td>
<td>Sorts eigenpairs by real and complex data types.</td>
</tr>
<tr>
<td>?lasrt2</td>
<td>s, d</td>
<td>Sorts numbers in increasing or decreasing order.</td>
</tr>
<tr>
<td>?stegr2</td>
<td>s, d</td>
<td>Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.</td>
</tr>
<tr>
<td>?stegr2a</td>
<td>s, d</td>
<td>Computes selected eigenvalues and initial representations needed for eigenvector computations.</td>
</tr>
<tr>
<td>?stegr2b</td>
<td>s, d</td>
<td>From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.</td>
</tr>
<tr>
<td>?stein2</td>
<td>s, d</td>
<td>Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.</td>
</tr>
<tr>
<td>?dbtf2</td>
<td>s, d, c, z</td>
<td>Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).</td>
</tr>
<tr>
<td>?dbtrf</td>
<td>s, d, c, z</td>
<td>Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).</td>
</tr>
<tr>
<td>?dttrf</td>
<td>s, d, c, z</td>
<td>Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).</td>
</tr>
<tr>
<td>?dttrsfl</td>
<td>s, d, c, z</td>
<td>Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.</td>
</tr>
<tr>
<td>?pttrsv</td>
<td>s, d, c, z</td>
<td>Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the LDLH factorization computed by ?pttrf.</td>
</tr>
<tr>
<td>?steqr2</td>
<td>s, d</td>
<td>Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.</td>
</tr>
<tr>
<td>?trmvt</td>
<td>s, d, c, z</td>
<td>Performs matrix-vector operations.</td>
</tr>
<tr>
<td>pilaenv</td>
<td>NA</td>
<td>Returns the positive integer value of the logical blocking size.</td>
</tr>
<tr>
<td>pilaenvx</td>
<td>NA</td>
<td>Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.</td>
</tr>
<tr>
<td>pjlaenv</td>
<td>NA</td>
<td>Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.</td>
</tr>
</tbody>
</table>

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
**p?lacgv**

*Conjugates a complex vector.*

**Syntax**

```c
void pclacgv (MKL_INT *n, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
void pzlacgv (MKL_INT *n, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?lacgv` function conjugates a complex vector `sub(X)` of length `n`, where `sub(X)` denotes `X(ix, jx:jx+n-1)` if `incx = m_x`, and `X(ix:ix+n-1, jx)` if `incx = 1`.

**Input Parameters**

- `n` (global) The length of the distributed vector `sub(X)`.
- `x` (local). Pointer into the local memory to an array of size `lld_x * LOCc(n_x)`. On entry the vector to be conjugated `x[i] = X(ix+(jx-1)*m_x+i*incx)` if `incx = m_x`, and `x[i] = x[ix+jx+(i-1)*incx]` if `incx = 1`.
- `ix` (global) The row index in the global matrix `X` indicating the first row of `sub(X)`.
- `jx` (global) The column index in the global matrix `X` indicating the first column of `sub(X)`.
- `descx` (global and local) Array of size `dlen_ = 9`. The array descriptor for the distributed matrix `X`.
- `incx` (global) The global increment for the elements of `X`. Only two values of `incx` are supported in this version, namely 1 and `m_x`. `incx` must not be zero.

**Output Parameters**

- `x` (local).
  On exit, the local pieces of conjugated distributed vector `sub(X)`.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?max1**

*Finds the index of the element whose real part has maximum absolute value (similar to the Level 1 PBLAS `p?amax`, but using the absolute value to the real part).*
Syntax

void pcmax1 (MKL_INT *n, MKL_Complex8 *amax, MKL_INT *indx, MKL_Complex8 *x,
MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);

void pzmax1 (MKL_INT *n, MKL_Complex16 *amax, MKL_INT *indx, MKL_Complex16 *x,
MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);

Include Files
• mkl_scalapack.h

Description
The \texttt{p?max1} function computes the global index of the maximum element in absolute value of a distributed vector \texttt{sub(X)}. The global index is returned in \texttt{indx} and the value is returned in \texttt{amax}, where \texttt{sub(X)} denotes \texttt{X(ix:ix}+\texttt{n-1,jx:} if \texttt{incx = 1}, \texttt{X(ix,jx:jx}+\texttt{n-1)} if \texttt{incx = m_x}.

Input Parameters

\begin{itemize}
\item \texttt{n} (global). The number of components of the distributed vector \texttt{sub(X)}. \texttt{n} \geq 0.
\item \texttt{x} (local) Pointer into the local memory to an array of size \texttt{lld_x} \times \texttt{LOCc} \texttt{(jx}+\texttt{n-1). On entry this array contains the local pieces of the distributed vector \texttt{sub(X)}.
\item \texttt{ix} (global) The row index in the global matrix \texttt{X} indicating the first row of \texttt{sub(X)}.
\item \texttt{jx} (global) The column index in the global matrix \texttt{X} indicating the first column of \texttt{sub(X)}.
\item \texttt{descx} (global and local) Array of size \texttt{dlen__. The array descriptor for the distributed matrix \texttt{X}.
\item \texttt{incx} (global). The global increment for the elements of \texttt{X}. Only two values of \texttt{incx} are supported in this version, namely \texttt{1} and \texttt{m_x}. \texttt{incx} must not be zero.
\end{itemize}

Output Parameters

\begin{itemize}
\item \texttt{amax} (global output). The absolute value of the largest entry of the distributed vector \texttt{sub(X)} only in the scope of \texttt{sub(X)}.
\item \texttt{indx} (global output). The global index of the element of the distributed vector \texttt{sub(X)} whose real part has maximum absolute value.
\end{itemize}

See Also

\texttt{Overview} for details of ScaLAPACK array descriptor structures and related notations.

\texttt{pilaver}

Returns the ScaLAPACK version.

Syntax

void pilaver (MKL_INT* vers_major, MKL_INT* vers_minor, MKL_INT* vers_patch);
Include Files

- mkl_scalapack.h

Description

This function returns the ScaLAPACK version.

Output Parameters

- vers_major
  Return the ScaLAPACK major version.
- vers_minor
  Return the ScaLAPACK minor version from the major version.
- vers_patch
  Return the ScaLAPACK patch version from the minor version.

pmpcol

Finds the collaborators of a process.

Syntax

```c
void pmpcol(MKL_INT* myproc, MKL_INT* nprocs, MKL_INT* iill, MKL_INT* needil, MKL_INT* neediu, MKL_INT* pmyils, MKL_INT* pmyius, MKL_INT* colbrt, MKL_INT* frstcl, MKL_INT* lastcl);
```

Include Files

- mkl_scalapack.h

Description

Using the output from pmpim2 and given the information on eigenvalue clusters, pmpcol finds the collaborators of myproc.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

- myproc
  The processor number, 0 ≤ myproc < nprocs.
- nprocs
  The total number of processors available.
- iill
  The index of the leftmost eigenvalue in the eigenvalue cluster.
- needil
  The leftmost position in the eigenvalue cluster needed by myproc.
- neediu
  The rightmost position in the eigenvalue cluster needed by myproc.
For each processor $p$, $0 < p \leq \text{nprocs}$, $p\text{myils}[p-1]$ is the index of the first eigenvalue in the eigenvalue cluster to be computed. $p\text{myils}[p-1]$ equals zero if $p$ stays idle.

$p\text{myius}$ array

For each processor $p$, $p\text{myius}[p-1]$ is the index of the last eigenvalue in the eigenvalue cluster to be computed. $p\text{myius}[p-1]$ equals zero if $p$ stays idle.

**OUTPUT Parameters**

$\text{colbrt}$

Non-zero if $\text{myproc}$ collaborates.

$\text{frstcl, lastcl}$

First and last collaborator of $\text{myproc}$.

$\text{myproc}$ collaborates with:

$\text{frstcl}, ..., \text{myproc}-1, \text{myproc}+1, ..., \text{lastcl}$

If $\text{myproc} = \text{frstcl}$, there are no collaborators on the left. If $\text{myproc} = \text{lastcl}$, there are no collaborators on the right.

If $\text{frstcl} = 0$ and $\text{lastcl} = \text{nprocs}-1$, then $\text{myproc}$ collaborates with everybody

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**pmpim2**

*Computes the eigenpair range assignments for all processes.*

**Syntax**

```c
void pmpim2(MKL_INT* il, MKL_INT* iu, MKL_INT* nprocs, MKL_INT* pmyils, MKL_INT* pmyius);
```

**Include Files**

- mkl_scalapack.h

**Description**

$pmpim2$ is the scheduling function. It computes for all processors the eigenpair range assignments.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
Input Parameters

*il, iu*
The range of eigenpairs to be computed.

*nprocs*
The total number of processors available.

Output Parameters

*pmyils*
array
For each processor *p*, *pmyils*[p-1] is the index of the first eigenvalue in a cluster to be computed.
*pmyils*[p-1] equals zero if *p* stays idle.

*pmyius*
array
For each processor *p*, *pmyius*[p-1] is the index of the last eigenvalue in a cluster to be computed.
*pmyius*[p-1] equals zero if *p* stays idle.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**?combamax1**

*Finds the element with maximum real part absolute value and its corresponding global index.*

**Syntax**

```c
void ccombamax1 (MKL_Complex8 *v1 , MKL_Complex8 *v2 );
void zcombamax1 (MKL_Complex16 *v1 , MKL_Complex16 *v2 );
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `?combamax1` function finds the element having maximum real part absolute value as well as its corresponding global index.

**Input Parameters**

`v1` (local)

`v2` (local)

**Output Parameters**

v1  (local).

The first maximum absolute value element and its global index.


**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?sum1**

Forms the 1-norm of a complex vector similar to Level 1 PBLAS p?asum, but using the true absolute value.

**Syntax**

void pscsum1 (MKL_INT *n, float *asum, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);

void pdzsum1 (MKL_INT *n, double *asum, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);

**Include Files**

- mkl_scalapack.h

**Description**

The p?sum1 function returns the sum of absolute values of a complex distributed vector sub(x) in asum, where sub(x) denotes X(ix:ix+n-1, jx:jx), if incx = 1, X(ix:ix, jx:jx+n-1), if incx = m_x.

Based on p?asum from the Level 1 PBLAS. The change is to use the 'genuine' absolute value.

**Input Parameters**

n  (global). The number of components of the distributed vector sub(x). n ≥ 0.

x  (local )

Pointer into the local memory to an array of size lld_x * LOCc(jx+n-1). This array contains the local pieces of the distributed vector sub(X).

ix  (global) The row index in the global matrix X indicating the first row of sub(X).

jx  (global) The column index in the global matrix X indicating the first column of sub(X)

descx  (local) Array of size dien_ =9. The array descriptor for the distributed matrix X.

incx  (global) The global increment for the elements of X. Only two values of incx are supported in this version, namely 1 and m_x.
Output Parameters

\texttt{asum} \quad (\text{(local)})

The sum of absolute values of the distributed vector sub\((X)\) only in its scope.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{p\texttt{d}dbtrsv}

Computes an LU factorization of a general triangular matrix with no pivoting. The function is called by \texttt{p\texttt{d}dbtrs}.

Syntax

\begin{verbatim}
void psdbtrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, float *a, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);
void pddbtrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, double *a, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);
void pcdbtrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzdbtrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *bwl, MKL_INT *bwu, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
\end{verbatim}

Include Files

- mkl_scalapack.h

Description

The \texttt{p\texttt{d}dbtrsv} function solves a banded triangular system of linear equations
\begin{align*}
A(1:n, ja:ja+n-1) \times X &= B(ib:ib+n-1, 1:nrhs) \quad \text{(for real flavors)}; \\
A(1:n, ja:ja+n-1)^T \times X &= B(ib:ib+n-1, 1:nrhs) \quad \text{(for complex flavors)},
\end{align*}
where \(A(1:n, ja:ja+n-1)\) is a banded triangular matrix factor produced by the Gaussian elimination code of \texttt{p\texttt{d}dbtrf} and is stored in \(A(1:n, ja:ja+n-1)\) and \(af\). The matrix stored in \(A(1:n, ja:ja+n-1)\) is either upper or lower triangular according to \texttt{uplo}, and the choice of solving \(A(1:n, ja:ja+n-1)\) or \((A(1:n, ja:ja+n-1))^T\) is dictated by the user by the parameter \texttt{trans}.

The function \texttt{p\texttt{d}dbtrf} must be called first.
Input Parameters

**uplo**

If `uplo='U'`, the upper triangle of \( A(1:n, ja:ja+n-1) \) is stored, if `uplo = 'L'`, the lower triangle of \( A(1:n, ja:ja+n-1) \) is stored.

**trans**

If `trans = 'N'`, solve with \( A(1:n, ja:ja+n-1) \), if `trans = 'C'`, solve with conjugate transpose \( A(1:n, ja:ja+n-1) \).

**n**

The order of the distributed submatrix \( A; (n \geq 0) \).

**bwl**

Number of subdiagonals. \( 0 \leq bwl \leq n-1 \).

**bwu**

Number of subdiagonals. \( 0 \leq bwu \leq n-1 \).

**nrhs**

The number of right-hand sides; the number of columns of the distributed submatrix \( B \) \( (nrhs \geq 0) \).

**a**

Pointer into the local memory to an array of size \( lld_a * LOCc(ja+n-1) \), where \( lld_a \geq (bwl+bwu+1) \). On entry, this array contains the local pieces of the \( n \)-by-\( n \) unsymmetric banded distributed Cholesky factor \( L \) or \( L^T \), represented in global \( A \) as \( A(1:n, ja:ja+n-1) \). This local portion is stored in the packed banded format used in LAPACK. See the Application Notes below and the ScaLAPACK manual for more detail on the format of distributed matrices.

**ja**

The index in the global matrix \( A \) that points to the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

**desca**

Array of size \( dlen_ \).

If 1d type \( (dtype_a = 501 \text{ or } 502) \), \( dlen \geq 7 \);

If 2d type \( (dtype_a = 1) \), \( dlen \geq 9 \). The array descriptor for the distributed matrix \( A \). Contains information of mapping of \( A \) to memory.

**b**

Pointer into the local memory to an array of local lead dimension \( lld_b \geq nb \). On entry, this array contains the local pieces of the right-hand sides \( B(ib:ib+n-1, 1:nrhs) \).

**ib**

The row index in the global matrix \( B \) that points to the first row of the matrix to be operated on (which may be either all of \( B \) or a submatrix of \( B \)).

**descb**

Array of size \( dlen_ \).

If 1d type \( (dtype_b = 502) \), \( dlen \geq 7 \);

If 2d type \( (dtype_b = 1) \), \( dlen \geq 9 \). The array descriptor for the distributed matrix \( B \). Contains information of mapping \( B \) to memory.

**laf**

Size of user-input auxiliary fill-in space \( af \).
If $\text{laf}$ is not large enough, an error code is returned and the minimum acceptable size will be returned in $\text{af}[0]$.

**work**

(local).

Temporary workspace. This space may be overwritten in between function calls.

*work* must be the size given in *lwork*.

**lwork**

(local or global)

Size of user-input workspace *work*. If *lwork* is too small, the minimal acceptable size will be returned in *work[0]* and an error code is returned.

$lwork \geq \max(bwl, bwu) \times \text{nrhs}$.

### Output Parameters

**$a$**

(local).

This local portion is stored in the packed banded format used in LAPACK. Please see the ScaLAPACK manual for more detail on the format of distributed matrices.

**$b$**

On exit, this contains the local piece of the solutions distributed matrix $X$.

**$af$**

(auxiliary fill-in space. The fill-in space is created in a call to the factorization function $p\text{?dbtrf}$ and is stored in $af$. If a linear system is to be solved using $p\text{?dbtrf}$ after the factorization function, $af$ must not be altered after the factorization.)

On exit, *work[0]* contains the minimal *lwork*.

**info**

(local).

If *info* = 0, the execution is successful.

$< 0$: If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $\text{info} = -(i \times 100 + j)$, if the $i$-th argument is a scalar and had an illegal value, then $\text{info} = -i$.

### See Also

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?dttrs**

Computes an LU factorization of a general band matrix, using partial pivoting with row interchanges. The function is called by $p\text{?dttrs}$.

**Syntax**

```c
void psdttrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, float *dl, float *d, float *du, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *iaf, float *work, MKL_INT *iwork, MKL_INT *info);
```
void pddttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, double *dl, double *d, double *du, MKL_INT *ja, MKL_INT *descb, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);

void pcdttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzdtttrs (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?dttrsv function solves a tridiagonal triangular system of linear equations

\[ A(1:n, ja:ja+n-1)X = B(ib:ib+n-1, 1:nrhs) \]

\[ A(1:n, ja:ja+n-1)^T X = B(ib:ib+n-1, 1:nrhs) \] for real flavors; \( A(1:n, ja:ja+n-1)^H X = B(ib:ib+n-1, 1:nrhs) \) for complex flavors,

where \( A(1:n, ja:ja+n-1) \) is a tridiagonal matrix factor produced by the Gaussian elimination code of p?dttrf and is stored in \( A(1:n, ja:ja+n-1) \) and \( af \). The matrix stored in \( A(1:n, ja:ja+n-1) \) is either upper or lower triangular according to \( uplo \), and the choice of solving \( A(1:n, ja:ja+n-1) \) or \( A(1:n, ja:ja+n-1)^T \) is dictated by the user by the parameter \( trans \).

The function p?dttrf must be called first.

Input Parameters

uplo

(global)

If \( uplo = 'U' \), the upper triangle of \( A(1:n, ja:ja+n-1) \) is stored,

if \( uplo = 'L' \), the lower triangle of \( A(1:n, ja:ja+n-1) \) is stored.

trans

(global)

If \( trans = 'N' \), solve with \( A(1:n, ja:ja+n-1) \),

if \( trans = 'C' \), solve with conjugate transpose \( A(1:n, ja:ja+n-1) \).

n

(global) The order of the distributed submatrix \( A; (n \geq 0) \).

nrhs

(global) The number of right-hand sides; the number of columns of the distributed submatrix \( B(ib:ib+n-1, 1:nrhs) \). \( (nrhs \geq 0) \).

dl

(local).

Pointer to local part of global vector storing the lower diagonal of the matrix.

Globally, \( dl[0] \) is not referenced, and \( dl \) must be aligned with \( d \).
Must be of size \( \geq nb_a \).

\( d \)

(local).

Pointer to local part of global vector storing the main diagonal of the matrix.

\( du \)

(local).

Pointer to local part of global vector storing the upper diagonal of the matrix.

Globally, \( du[n-1] \) is not referenced, and \( du \) must be aligned with \( d \).

\( ja \)

(global) The index in the global matrix \( A \) that points to the start of the matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( desca \)

(global and local) array of size \( dlen_\).

if 1d type (\( dtype_a = 501 \) or 502), \( dlen \geq 7 \);

if 2d type (\( dtype_a = 1 \)), \( dlen \geq 9 \).

The array descriptor for the distributed matrix \( A \). Contains information of mapping of \( A \) to memory.

\( b \)

(local)

Pointer into the local memory to an array of local lead dimension \( lld_b : \geq nb \). On entry, this array contains the local pieces of the right-hand sides \( B(ib : ib+n-1, 1 : nrhs) \).

\( ib \)

(global) The row index in the global matrix \( B \) that points to the first row of the matrix to be operated on (which may be either all of \( B \) or a submatrix of \( B \)).

\( descb \)

(global and local) array of size \( dlen_\).

if 1d type (\( dtype_b = 502 \)), \( dlen \geq 7 \);

if 2d type (\( dtype_b = 1 \)), \( dlen \geq 9 \).

The array descriptor for the distributed matrix \( B \). Contains information of mapping \( B \) to memory.

\( laf \)

(local).

Size of user-input auxiliary fill-in space \( af \).

\( laf \geq 2*(nb+2) \). If \( laf \) is not large enough, an error code is returned and the minimum acceptable size will be returned in \( af[0] \).

\( work \)

(local).

Temporary workspace. This space may be overwritten in between function calls.

\( work \) must be the size given in \( lwork \).

\( lwork \)

(local or global)

Size of user-input workspace \( work \). If \( lwork \) is too small, the minimal acceptable size will be returned in \( work[0] \) and an error code is returned.

\( lwork \geq 10*ncol+4*nrhs \).
Output Parameters

\(dl\)

On exit, this array contains information containing the factors of the matrix.

\(d\)

On exit, this array contains information containing the factors of the matrix.

Must be of size \(\geq nb_a\).

\(b\)

On exit, this contains the local piece of the solutions distributed matrix X.

\(af\)

Auxiliary fill-in space. The fill-in space is created in a call to the factorization function \(p?dttrf\) and is stored in \(af\). If a linear system is to be solved using \(p?dttrs\) after the factorization function, \(af\) must not be altered after the factorization.

\(work\)

On exit, \(work[0]\) contains the minimal \(lwork\).

\(info\)

If \(info=0\), the execution is successful.

if \(info<0\): If the \(i\)-th argument is an array and the \(j\)-th entry, indexed \(j-1\), had an illegal value, then \(info = -(i*100+j)\), if the \(i\)-th argument is a scalar and had an illegal value, then \(info = -i\).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\(p?gebal\)

Balances a general real/complex matrix.

Syntax

```
void psgebal(char* job, MKL_INT* n, float* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT* ihi, float* scale, MKL_INT* info);
void pdgebal(char* job, MKL_INT* n, double* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT* ihi, double* scale, MKL_INT* info);
void pcgebal(char* job, MKL_INT* n, complex float* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT* ihi, float* scale, MKL_INT* info);
void pzgebal(char* job, MKL_INT* n, complex double* a, MKL_INT* desca, MKL_INT* ilo, MKL_INT* ihi, double* scale, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

\(p?gebal\) balances a general real/complex matrix \(A\). This involves, first, permuting \(A\) by a similarity transformation to isolate eigenvalues in the first \(ilo\)-1 and last \(ihi+1\) to \(n\) elements on the diagonal; and second, applying a diagonal similarity transformation to rows and columns \(ilo\) to \(ihi\) to make the rows and columns as close in norm as possible. Both steps are optional.

Balancing may reduce the 1-norm of the matrix, and improve the accuracy of the computed eigenvalues and/or eigenvectors.
**Input Parameters**

\textit{job}  
(global )  
Specifies the operations to be performed on \( a \):
- = 'N': none: simply set \( ilo = 1, ihi = n, \text{scale}[i] = 1.0 \) for \( i = 0,\ldots,n-1 \);
- = 'P': permute only;
- = 'S': scale only;
- = 'B': both permute and scale.

\textit{n}  
(global )  
The order of the matrix \( A \) \((n \geq 0)\).

\textit{a}  
(local )  
Pointer into the local memory to an array of size \( \text{lld}_a \times \text{LOC}_{c(n)} \)  
This array contains the local pieces of global input matrix \( A \).

\textit{desca}  
(global and local) array of size \( \text{dlen}_c \)  
The array descriptor for the distributed matrix \( A \).

**OUTPUT Parameters**

\textit{a}  
On exit, \( a \) is overwritten by the balanced matrix \( A \).
If \( \text{job} = 'N' \), \( a \) is not referenced.
See Notes for further details.

\textit{ilo, ihi}  
(global )  
\( ilo \) and \( ihi \) are set to integers such that on exit matrix elements \( A(i,j) \) are zero if \( i > j \) and \( j = 1,\ldots,ilo-1 \) or \( i = ihi+1,\ldots,n \).
If \( \text{job} = 'N' \) or 'S', \( ilo = 1 \) and \( ihi = n \).

\textit{scale}  
(global )  
array of size \( n \).
Details of the permutations and scaling factors applied to \( a \). If \( pj \) is the index of the row and column interchanged with row and column \( j \) and \( dj \) is the scaling factor applied to row and column \( j \), then
\[
\text{scale}[j-1] = pj \quad \text{for } j = 1,\ldots,ilo-1, \ ihi+1,\ldots, n \]
\[
\text{scale}[j-1] = dj \quad \text{for } j = ilo,\ldots,ihi
\]
The order in which the interchanges are made is \( n \) to \( ihi+1 \), then \( 1 \) to \( ilo-1 \).

\textit{info}  
(global )  
= 0: successful exit.
< 0: if \( \text{info} = -i \), the \( i \)-th argument had an illegal value.

**Application Notes**
The permutations consist of row and column interchanges which put the matrix in the form
where $T_1$ and $T_2$ are upper triangular matrices whose eigenvalues lie along the diagonal. The column indices $ilo$ and $ihi$ mark the starting and ending columns of the submatrix $B$. Balancing consists of applying a diagonal similarity transformation $D^{-1}BD$ to make the 1-norms of each row of $B$ and its corresponding column nearly equal. The output matrix is

$$
\begin{pmatrix}
T_1 & XD & Y \\
0 & D^{-1}BD & D^{-1}Z \\
0 & 0 & T_2
\end{pmatrix}
$$

Information about the permutations $P$ and the diagonal matrix $D$ is returned in the vector $scale$.

See Also
Overview for details of ScalAPACK array descriptor structures and related notations.

**p?gebd2**

*Reduces a general rectangular matrix to real bidiagonal form by an orthogonal/unitary transformation (unblocked algorithm).*

**Syntax**

```c
void psgebd2 (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, float *taup, float *work, MKL_INT *iwork, MKL_INT *info);
void pdgebd2 (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, double *taup, double *work, MKL_INT *iwork, MKL_INT *info);
void pcgebd2 (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, MKL_Complex8 *taup, MKL_Complex8 *work, MKL_INT *iwork, MKL_INT *info);
void pzgebd2 (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, MKL_Complex16 *taup, MKL_Complex16 *work, MKL_INT *iwork, MKL_INT *info);
```

**Include Files**
- mkl_scalapack.h

**Description**

The `p?gebd2` function reduces a real/complex general $m$-by-$n$ distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$ to upper or lower bidiagonal form $B$ by an orthogonal/unitary transformation:

$$Q^*\text{sub}(A)*P = B.$$ 

If $m \geq n$, $B$ is the upper bidiagonal; if $m < n$, $B$ is the lower bidiagonal.
Input Parameters

\(m\)  
(global)  
The number of rows of the distributed matrix \(\text{sub}(A)\). \((m \geq 0)\).

\(n\)  
(global)  
The number of columns in the distributed matrix \(\text{sub}(A)\). \((n \geq 0)\).

\(a\)  
(local).  
Pointer into the local memory to an array of size \(\text{lld}_a \times \text{LOC}(ja+n-1)\).  
On entry, this array contains the local pieces of the general distributed matrix \(\text{sub}(A)\).

\(ia, ja\)  
(global)  
The row and column indices in the global matrix \(A\) indicating the first row and the first column of the matrix \(\text{sub}(A)\), respectively.

\(\text{desca}\)  
(global and local) array of size \(\text{dlen}_\text{a}\). The array descriptor for the distributed matrix \(A\).

\(\text{work}\)  
(local).  
This is a workspace array of size \(lwork\).

\(lwork\)  
(local or global)  
The size of the array \(\text{work}\).  
\(\text{lwork}\) is local input and must be at least \(lwork \geq \max(\text{mpa}0, \text{nqa}0)\),  
where \(nb = \text{mb}_a = \text{nb}_a\), \(\text{ioffa} = \text{mod}(\text{ia}-1, \text{nb})\),  
\(\text{irow} = \text{indxg2p}(\text{ia}, \text{nb}, \text{myrow}, \text{rsrc}_a, \text{nprow})\),  
\(\text{icol} = \text{indxg2p}(\text{ja}, \text{nb}, \text{mycol}, \text{csrc}_a, \text{npcol})\),  
\(\text{mpa}0 = \text{numroc}(m+\text{ioffa}, \text{nb}, \text{myrow}, \text{iarow}, \text{nprow})\),  
\(\text{nqa}0 = \text{numroc}(n+i\text{coff}a, \text{nb}, \text{mycol}, \text{iacol}, \text{ncol})\).  
\text{indxg2p} and \text{numroc} are ScaLAPACK tool functions; \text{myrow}, \text{mycol}, \text{nprow}, and \text{npcol} can be determined by calling the function \text{blacs_gridinfo}.

If \(lwork = -1\), then \(\text{lwork}\) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\(a\)  
(local).  
On exit, if \(m \geq n\), the diagonal and the first superdiagonal of \(\text{sub}(A)\) are overwritten with the upper bidiagonal matrix \(B\); the elements below the diagonal, with the array \(\text{tauq}\), represent the orthogonal/unitary matrix \(Q\) as a product of elementary reflectors, and the elements above the first superdiagonal, with the array \(\text{taup}\), represent the orthogonal matrix \(P\) as a product of elementary reflectors. If \(m < n\), the diagonal and the first subdiagonal are overwritten with the lower bidiagonal matrix \(B\); the elements below the first subdiagonal, with the array \(\text{tauq}\), represent the
orthogonal/unitary matrix Q as a product of elementary reflectors, and the
elements above the diagonal, with the array $\text{taup}$, represent the orthogonal
matrix $P$ as a product of elementary reflectors. See Applications Notes
below.

$d$

(local)

Array of size $\text{LOCc}(ja+\min(m,n)-1)$ if $m \geq n$; $\text{LOCr}(ia+\min(m,n)-1)$
otherwise. The distributed diagonal elements of the bidiagonal matrix $B$:

$d[i] = A(i+1,i+1)$, $i=0, 1, ..., \text{size}(d) - 1$.

$d$ is tied to the distributed matrix $A$.

e

(local)

Array of size $\text{LOCc}(ja+\min(m,n)-1)$ if $m \geq n$; $\text{LOCr}(ia+\min(m,n)-2)$
otherwise. The distributed diagonal elements of the bidiagonal matrix $B$:

if $m \geq n$, $e[i] = A(i+2,i+1)$ for $i = 0, 1, ..., m-2$.

$e$ is tied to the distributed matrix $A$.

tauq

(local).

Array of size $\text{LOCc}(ja+\min(m,n)-1)$. The scalar factors of the elementary
reflectors which represent the orthogonal/unitary matrix $Q$. $\text{tauq}$ is tied to
the distributed matrix $A$.

taup

(local).

Array of size $\text{LOCr}(ia+\min(m,n)-1)$. The scalar factors of the elementary
reflectors which represent the orthogonal/unitary matrix $P$. $\text{taup}$ is tied to
the distributed matrix $A$.

work

On exit, $\text{work}[0]$ returns the minimal and optimal $lwork$. 

info

(local)

If $\text{info} = 0$, the execution is successful.

if $\text{info} < 0$: If the $i$-th argument is an array and the $j$-th entry, indexed

$j-1$, had an illegal value, then $\text{info} = -(i*100+j)$, if the $i$-th argument is a

scalar and had an illegal value, then $\text{info} = -i$.

Application Notes

The matrices $Q$ and $P$ are represented as products of elementary reflectors:

If $m \geq n$,

$Q = H(1)H(2)\ldots H(n)$, and $P = G(1)G(2)\ldots G(n-1)$

Each $H(i)$ and $G(i)$ has the form:

$H(i) = I - \text{tauq}v^*v'$, and $G(i) = I - \text{taup}u^*u'$,

where $\text{tauq}$ and $\text{taup}$ are real/complex scalars, and $v$ and $u$ are real/complex vectors. $v(1:i-1) = 0$, $v(i) = 1$, and $v(i+i:m)$ is stored on exit in

$A(ia+i-ia+m-1, ja+i-1)$;

$u(1:i) = 0$, $u(i+1) = 1$, and $u(i+2:n)$ is stored on exit in $A(ia+i-1, ja+i+1:ja+n-1)$;
tauq is stored in \( \text{tauq}[ja+i-2] \) and \( \text{taup} \) in \( \text{taup}[ia+i-2] \).

If \( m < n \),

\[
\begin{align*}
v(1: i) &= 0, \quad v(i+1) = 1, \quad \text{and} \quad v(i+2:m) \text{ is stored on exit in } A(ia+i+1: ia+m-1, ja+i-1); \\
u(1: i-1) &= 0, \quad u(i) = 1, \quad \text{and} \quad u(i+1:n) \text{ is stored on exit in } A(ia+i-1, ja+i: ja+n-1);
\end{align*}
\]

\( \tau \) is stored in \( \text{tauq}[ja+i-2] \) and \( \text{taup} \) in \( \text{taup}[ia+i-2] \).

The contents of sub(A) on exit are illustrated by the following examples:

\[
\begin{array}{c}
m = 6 \text{ and } n = 5 (m > n): \\
\begin{bmatrix}
v1 & v2 & v3 & v4 & v5 \\
v1 & v2 & v3 & v4 & d \\
v1 & v2 & v3 & e & d \\
v1 & d & e & u3 & u2 \\
v & d & e & u2 & u1 \\
v & d & u1 & u1 & u1
\end{bmatrix}
\end{array}
\]

where \( d \) and \( e \) denote diagonal and off-diagonal elements of \( B \), \( vi \) denotes an element of the vector defining \( H(i) \), and \( ui \) an element of the vector defining \( G(i) \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gehd2

Reduces a general matrix to upper Hessenberg form by an orthogonal/unitary similarity transformation (unblocked algorithm).

Syntax

void psgehd2 (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *iwork, MKL_INT *info);

void pdgehd2 (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *iwork, MKL_INT *info);

void pcgehd2 (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *iwork, MKL_INT *info);

void pzgehd2 (MKL_INT *n, MKL_INT *ilo, MKL_INT *ihi, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *iwork, MKL_INT *info);

Include Files

- mkl_scalapack.h

Description

The p?gehd2 function reduces a real/complex general distributed matrix sub(A) to upper Hessenberg form \( H \) by an orthogonal/unitary similarity transformation: \( Q' \)*sub(A)*Q = H, where sub(A) = A(ia+n-1 : ia +n-1, ja+n-1 : ja+n-1).
Input Parameters

\( n \) (global) The order of the distributed submatrix \( A \). \((n \geq 0)\).  

\( ilo, \text{ ihi} \) (global) It is assumed that the matrix \( \text{sub}(A) \) is already upper triangular in rows \( ia:ia+ilo-2 \) and \( ia+ihi:ia+n-1 \) and columns \( ja:ja+jlo-2 \) and \( ja+jhi:ja+n-1 \). See Application Notes for further information.  

If \( n \geq 0, 1 \leq ilo \leq ihi \leq n \); otherwise set \( ilo = 1, ihi = n \).

\( a \) (local).  

Pointer into the local memory to an array of size \( lld_a * \text{LOC}(ja+n-1) \).  

On entry, this array contains the local pieces of the \( n \)-by-\( n \) general distributed matrix \( \text{sub}(A) \) to be reduced.

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen\_ \). The array descriptor for the distributed matrix \( A \).

\( \text{work} \) (local).  

This is a workspace array of size \( lwork \).

\( lwork \) (local or global)  

The size of the array \( \text{work} \).  

\( lwork \) is local input and must be at least \( lwork \geq nb + \max( npa0, nb ) \), where \( nb = mb_a = nb_a, iroff_a = \text{mod}( ia-1, nb ), iarow = \text{indxg2p}( ia, nb, myrow, rsrq_a, nprow ), npa0 = \text{numroc}(ihi +iroff_a, nb, myrow, iarow, nprow ) \).  

\text{indxg2p} and \text{numroc} are ScALAPACK tool functions; \text{myrow, mycol, nprow, and npcol} can be determined by calling the function \text{blacs_gridinfo}.  

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\( a \) (local).  

On exit, the upper triangle and the first subdiagonal of sub\( (A) \) are overwritten with the upper Hessenberg matrix \( H \), and the elements below the first subdiagonal, with the array \( \text{tau} \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors. (see Application Notes below).

\( \text{tau} \) (local).  

Array of size \( \text{LOCc}(ja+n-2) \) The scalar factors of the elementary reflectors (see Application Notes below). Elements \( ja:ja+ilo-2 \) and \( ja+ihi:ja+n-2 \) of the global vector \( \text{tau} \) are set to zero. \( \text{tau} \) is tied to the distributed matrix \( A \).  

\( \text{work} \)  

On exit, \( \text{work}[0] \) returns the minimal and optimal \( lwork \).
If $info = 0$, the execution is successful.

if $info < 0$: If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*100+j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

Application Notes

The matrix $Q$ is represented as a product of $(ihi-ilo)$ elementary reflectors

$$ Q = H(ilo)*H(ilo+1)*...*H(ihi-1). $$

Each $H(i)$ has the form

$$ H(i) = I - \tau a^T v^T, $$

where $\tau$ is a real/complex scalar, and $v$ is a real/complex vector with $v(1:i)=0, v(i+1)=1$ and $v(i+1:n)=0; v(i+2:ihi)$ is stored on exit in $A(ia+ilo+ia+ihi-1, ia+ilo+i-2)$, and $\tau$ in $\tau[ja+ilo+i-3]$.

The contents of $A(ia:ia+n-1, ja:ja+n-1)$ are illustrated by the following example, with $n = 7, ilo = 2$ and $ihi = 6$:

<table>
<thead>
<tr>
<th>on entry</th>
<th>on exit</th>
</tr>
</thead>
<tbody>
<tr>
<td>a a a a a a a</td>
<td>a a h h h h a</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>a h h h h h</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>a v2 h h h h</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>v2 v3 h h h h</td>
</tr>
<tr>
<td>a a a a a a a</td>
<td>a v2 v3 v4 h h h</td>
</tr>
</tbody>
</table>

where $a$ denotes an element of the original matrix sub($A$), $h$ denotes a modified element of the upper Hessenberg matrix $H$, and $vi$ denotes an element of the vector defining $H(ja+ilo+i-2)$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?gelq2

Computes an LQ factorization of a general rectangular matrix (unblocked algorithm).

Syntax

```c
void psgelq2 (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgelq2 (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgelq2 (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgelq2 (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```
Include Files

- mkl_scalapack.h

Description

The \texttt{p?gelq2} function computes an LQ factorization of a real/complex distributed \( m \)-by-\( n \) matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = L*Q \).

Input Parameters

\( m \)

(global)

The number of rows of the distributed matrix \( \text{sub}(A) \). \( (m \geq 0) \).

\( n \)

(global)

The number of columns of the distributed matrix \( \text{sub}(A) \). \( (n \geq 0) \).

\( a \)

(local).

Pointer into the local memory to an array of size \( lld_a * \text{LOC}(ja+n-1) \).

On entry, this array contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \) which is to be factored.

\( ia, ja \)

(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of \( \text{sub}(A) \), respectively.

\( \text{desca} \)

(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \text{work} \)

(local).

This is a workspace array of size \( lwork \).

\( lwork \)

(local or global)

The size of the array \( \text{work} \).

\( lwork \) is local input and must be at least \( lwork \geq nq0 + \max(1, mp0) \), where

\[
\text{iroff} = \text{mod}(ia-1, mb_a), \quad \text{icoff} = \text{mod}(ja-1, nb_a),
\]

\[
\text{iarow} = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow),
\]

\[
\text{iacol} = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol),
\]

\[
mp0 = \text{numroc}(m+\text{iroff}, mb_a, myrow, iarow, nprow),
\]

\[
nq0 = \text{numroc}(n+\text{icoff}, nb_a, mycol, iacol, npcol),
\]

\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( \text{myrow, mycol, nprow, and npcol} \) can be determined by calling the function \text{blacs_gridinfo}.

If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters

\( a \)

(local).
On exit, the elements on and below the diagonal of \( \text{sub}(A) \) contain the \( m \) by \( \min(m,n) \) lower trapezoidal matrix \( L \) (\( L \) is lower triangular if \( m \leq n \)); the elements above the diagonal, with the array \( \text{tau} \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\[\text{tau}\] (local).

Array of size \( \text{LOCr}(\text{ia}+\min(m,n)-1) \). This array contains the scalar factors of the elementary reflectors. \( \text{tau} \) is tied to the distributed matrix \( A \).

\[\text{work}\] On exit, \( \text{work}[0] \) returns the minimal and optimal \( lwork \).

\[\text{info}\] (local) If \( \text{info} = 0 \), the execution is successful. If \( \text{info} < 0 \): If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = - (i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors

\[ Q = H(ia+k-1)*H(ia+k-2)*..*H(ia) \text{ for real flavors, } Q = (H(ia+k-1))^H*(H(ia+k-2))^H...*(H(ia))^H \text{ for complex flavors, } \]

where \( k = \min(m,n) \).

Each \( H(i) \) has the form

\[ H(i) = I - \tau_i v_i v_i^T \]

where \( \tau_i \) is a real/complex scalar, and \( v_i \) is a real/complex vector with \( v(1:i-1) = 0 \) and \( v(i) = 1; v(i+1:n) \) (for real flavors) or \( \text{conjg}(v(i+1:n)) \) (for complex flavors) is stored on exit in \( A(ia+i-1,ja+i:ja+n-1) \), and \( \tau_i \) in \( \text{tau}[ia+i-2] \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?geql2

Computes a QL factorization of a general rectangular matrix (unblocked algorithm).

Syntax

void psggeql2 (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgeql2 (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgeql2 (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgeql2 (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
Include Files
• mkl_scalapack.h

Description
The pageql2 function computes a QL factorization of a real/complex distributed \( m \)-by-\( n \) matrix \( \text{sub}(A) = A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1) = Q^*L \).

Input Parameters
\( m \) (global)
The number of rows in the distributed matrix \( \text{sub}(A) \). \( (m \geq 0) \).
\( n \) (global)
The number of columns in the distributed matrix \( \text{sub}(A) \). \( (n \geq 0) \).
a (local).
Pointer into the local memory to an array of size \( lld_a \times \text{LOC}_c(\text{ja}+n-1) \).
On entry, this array contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \) which is to be factored.
\( \text{ia}, \text{ja} \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of \( \text{sub}(A) \), respectively.
desca (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).
work (local).
This is a workspace array of size \( lwork \).
lwork (local or global)
The size of the array \( \text{work} \).
lwork is local input and must be at least \( lwork \geq mp0 + \max(1, nq0) \), where \( iroff = \text{mod}(\text{ia}-1, \text{mb}_a), \text{icoff} = \text{mod}(\text{ja}-1, \text{nb}_a) \), \( \text{irow} = \text{indxg2p}(\text{ia}, \text{mb}_a, \text{myrow}, \text{rsrc}_a, \text{nprow}) \), \( \text{icol} = \text{indxg2p}(\text{ja}, \text{nb}_a, \text{mycol}, \text{csrc}_a, \text{npcol}) \), \( mp0 = \text{numroc}(m+iroff, \text{mb}_a, \text{myrow}, \text{irow}, \text{nprow}) \), \( nq0 = \text{numroc}(n+icoff, \text{nb}_a, \text{mycol}, \text{icol}, \text{npcol}) \), \( \text{indxg2p} \) and \( \text{numroc} \) are ScALAPACK tool functions; \( \text{myrow}, \text{mycol}, \text{nprow}, \) and \( \text{npcol} \) can be determined by calling the function \text{blacs_gridinfo}.
If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \text{pxerbla}.

Output Parameters
a (local).
On exit,
if \( m \geq n \), the lower triangle of the distributed submatrix \( A((ia+m-n),ia+m-1,ja:ja+n-1) \) contains the \( n \)-by-\( n \) lower triangular matrix \( L \);  
if \( m \leq n \), the elements on and below the \((n-m)\)-th superdiagonal contain the \( m \)-by-\( n \) lower trapezoidal matrix \( L \); the remaining elements, with the array \( tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\[
\text{tau} (\text{local}).
\]

Array of size \( \text{LOCc}(ja+n-1) \). This array contains the scalar factors of the elementary reflectors. \( \text{tau} \) is tied to the distributed matrix \( A \).

\[
\text{work} (\text{local}).
\]

On exit, \( \text{work}[0] \) returns the minimal and optimal \( \text{lwork} \).

\[
\text{info} (\text{local}).
\]

If \( \text{info} = 0 \), the execution is successful. If \( \text{info} < 0 \): If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

Application Notes

The matrix \( Q \) is represented as a product of elementary reflectors
\[
Q = H(ja+k-1)*...*H(ja+1)*H(ja), \text{ where } k = \min(m,n).
\]
Each \( H(i) \) has the form
\[
H(i) = I - \tau v^*v',
\]
where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(m-k+i+1:m) = 0 \) and \( v(m-k+i) = 1 \); \( v(1:m-k+i-1) \) is stored on exit in \( A((ia:ia+m-k+i-2,ja+n-k+i-1)) \), and \( \tau \) in \( \text{tau}[ja+n-k+i-2] \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\text{p?geqr2}  
Computes a QR factorization of a general rectangular matrix (unblocked algorithm).

Syntax

\[
\begin{align*}
\text{void } \text{psgeqr2} (\text{MKL_INT } &m, \text{MKL_INT } n, \text{float } &a, \text{MKL_INT } &ia, \text{MKL_INT } &ja, \text{MKL_INT } &desca, \text{float } &tau, \text{float } &work, \text{MKL_INT } &lwork, \text{MKL_INT } &info );
\text{void } \text{pdgeqr2} (\text{MKL_INT } &m, \text{MKL_INT } n, \text{double } &a, \text{MKL_INT } &ia, \text{MKL_INT } &ja, \text{MKL_INT } &desca, \text{double } &tau, \text{double } &work, \text{MKL_INT } &lwork, \text{MKL_INT } &info );
\text{void } \text{pcgeqr2} (\text{MKL_INT } &m, \text{MKL_INT } n, \text{MKL_Complex8 } &a, \text{MKL_INT } &ia, \text{MKL_INT } &ja, \text{MKL_INT } &desca, \text{MKL_Complex8 } &tau, \text{MKL_Complex8 } &work, \text{MKL_INT } &lwork, \text{MKL_INT } &info );
\text{void } \text{pzgeqr2} (\text{MKL_INT } &m, \text{MKL_INT } n, \text{MKL_Complex16 } &a, \text{MKL_INT } &ia, \text{MKL_INT } &ja, \text{MKL_INT } &desca, \text{MKL_Complex16 } &tau, \text{MKL_Complex16 } &work, \text{MKL_INT } &lwork, \text{MKL_INT } &info );
\end{align*}
\]
Include Files
- mkl_scalapack.h

Description
The `pgeqr2` function computes a QR factorization of a real/complex distributed \( m \)-by-\( n \) matrix
\[
\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = Q^*R.
\]

Input Parameters

- \( m \) (global)
The number of rows in the distributed matrix \( \text{sub}(A) \). \((m \geq 0)\).

- \( n \) (global)
The number of columns in the distributed matrix \( \text{sub}(A) \). \((n \geq 0)\).

- \( a \) (local)
Pointer into the local memory to an array of size \( \text{lid}_a \ast \text{LOC}(ja+n-1) \).
On entry, this array contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \) which is to be factored.

- \( ia, ja \) (global)
The row and column indices in the global matrix \( A \) indicating the first row and the first column of \( \text{sub}(A) \), respectively.

- \( \text{desca} \) (global and local) array of size \( \text{dlen}_- \). The array descriptor for the distributed matrix \( A \).

- \( \text{work} \) (local)
This is a workspace array of size \( \text{lwork} \).

- \( \text{lwork} \) (local or global)
The size of the array \( \text{work} \).
\( \text{lwork} \) is local input and must be at least \( \text{lwork} \geq mp_0 + \max(1, nq_0) \), where \( \text{iroff} = \text{mod}(ia-1, \text{mb}_a) \), \( \text{icoff} = \text{mod}(ja-1, \text{nb}_a) \),
\( \text{iarow} = \text{indxg2p}(ia, \text{mb}_a, \text{myrow}, \text{rsrc}_a, \text{nprow}) \),
\( \text{iacol} = \text{indxg2p}(ja, \text{nb}_a, \text{mycol}, \text{csrc}_a, \text{npcol}) \),
\( mp_0 = \text{numroc}(m+\text{iroff}, \text{mb}_a, \text{myrow}, \text{iarow}, \text{nprow}) \),
\( nq_0 = \text{numroc}(n+\text{icoff}, \text{nb}_a, \text{mycol}, \text{iacol}, \text{npcol}) \).
\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( \text{myrow}, \text{mycol}, \text{nprow} \), and \( \text{npcol} \) can be determined by calling the function \( \text{blacs_gridinfo} \).
If \( \text{lwork} = -1 \), then \( \text{lwork} \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \( \text{pxerbla} \).

Output Parameters

- \( a \) (local).


On exit, the elements on and above the diagonal of sub$(A)$ contain the \(\min(m,n)\) by \(n\) upper trapezoidal matrix \(R\) (\(R\) is upper triangular if \(m\geq n\)); the elements below the diagonal, with the array \(\tau\), represent the orthogonal/unitary matrix \(Q\) as a product of elementary reflectors (see Application Notes below).

\textbf{tau} (local).

Array of size \(\text{LOCc}(ja+\min(m,n)-1)\). This array contains the scalar factors of the elementary reflectors. \(\tau\) is tied to the distributed matrix \(A\).

\textbf{work} (local)

On exit, \(\text{work}[0]\) returns the minimal and optimal \(lwork\).

\textbf{info} (local)

If \(\text{info} = 0\), the execution is successful. If \(\text{info} < 0\):

If the \(i\)-th argument is an array and the \(j\)-th entry, indexed \(j-1\), had an illegal value, then \(\text{info} = -(i*100+j)\),

if the \(i\)-th argument is a scalar and had an illegal value, then \(\text{info} = -i\).

\textbf{Application Notes}

The matrix \(Q\) is represented as a product of elementary reflectors

\[Q = H(ja)*H(ja+1)*\ldots*H(ja+k-1), \text{ where } k = \min(m,n).\]

Each \(H(i)\) has the form

\[H(j) = I - \tau v^*v',\]

where \(\tau\) is a real/complex scalar, and \(v\) is a real/complex vector with \(v(1:i-1) = 0\) and \(v(i) = 1\); \(v(i+1:m)\) is stored on exit in \(A(ia+i:ia+m-1, ja+i-1)\), and \(\tau\) in \(\tau[a+j-2]\).

\textbf{See Also}

\textbf{Overview} for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?gerq2}

\textit{Computes an RQ factorization of a general rectangular matrix (unblocked algorithm).}

\textbf{Syntax}

\begin{verbatim}
void psgerq2 (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdgerq2 (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcgerq2 (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzgerq2 (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
\end{verbatim}

\textbf{Include Files}

- mkl_scalapack.h
The *pgerq2* function computes an $RQ$ factorization of a real/complex distributed $m$-by-$n$ matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) = R*Q$.

**Input Parameters**

$m$ (global) The number of rows in the distributed matrix $\text{sub}(A)$. ($m \geq 0$).

$n$ (global) The number of columns in the distributed matrix $\text{sub}(A)$. ($n \geq 0$).

$a$ (local). Pointer into the local memory to an array of size $\text{lld}_a * \text{LOC}(ja+n-1)$.

On entry, this array contains the local pieces of the $m$-by-$n$ distributed matrix $\text{sub}(A)$ which is to be factored.

$ia, ja$ (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of $\text{sub}(A)$, respectively.

$\text{desca}$ (global and local) array of size $\text{dlen}_a$. The array descriptor for the distributed matrix $A$.

$\text{work}$ (local). This is a workspace array of size $\text{lwork}$.

$lwork$ (local or global) The size of the array $\text{work}$.

$lwork$ is local input and must be at least $lwork_0 = \text{nq} + \max(1, mp_0)$, where

\begin{align*}
\text{iroff} & = \text{mod}(ia-1, \text{mb}_a), \\
\text{icoff} & = \text{mod}(ja-1, \text{nb}_a), \\
\text{iarow} & = \text{indxg2p}(ia, \text{mb}_a, \text{myrow}, \text{rsrc}_a, \text{nprow}), \\
\text{iacol} & = \text{indxg2p}(ja, \text{nb}_a, \text{mycol}, \text{csrc}_a, \text{npcol}), \\
\text{mp} & = \text{numroc}(m+\text{iroff}, \text{mb}_a, \text{myrow}, \text{iarow}, \text{nprow}), \\
\text{nq} & = \text{numroc}(n+\text{icoff}, \text{nb}_a, \text{mycol}, \text{iacol}, \text{npcol}), \\
\text{indxg2p} \text{ and } \text{numroc} \text{ are ScALAPACK tool functions; } \text{myrow, mycol, nprow,} \\
\text{and } \text{npcol} \text{ can be determined by calling the function } \text{blacs_gridinfo}. \\
\end{align*}

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by *pxerbla*.

**Output Parameters**

$a$ (local).

On exit,

if $m \leq n$, the upper triangle of $A(ia+m-n:ia+m-1, ja:ja+n-1)$ contains the $m$-by-$m$ upper triangular matrix $R$;
if \( m \geq n \), the elements on and above the \((m-n)\)-th subdiagonal contain the \( m \)-by-\( n \) upper trapezoidal matrix \( R \); the remaining elements, with the array \( \tau \), represent the orthogonal/ unitary matrix \( Q \) as a product of elementary reflectors (see Application Notes below).

\( \tau \) (local).

Array of size \( \text{LOCr}(ia+m-1) \). This array contains the scalar factors of the elementary reflectors. \( \tau \) is tied to the distributed matrix \( A \).

\( \text{work} \) (local).

On exit, \( \text{work}[0] \) returns the minimal and optimal \( lwork \).

\( \text{info} \) (local)

If \( \text{info} = 0 \), the execution is successful.

If \( \text{info} < 0 \): If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

**Application Notes**

The matrix \( Q \) is represented as a product of elementary reflectors

\[
Q = H(ia)*H(ia+1)*...*H(ia+k-1) \quad \text{for real flavors},
\]

\[
Q = (H(ia))^H*(H(ia+1))^H*...*(H(ia+k-1))^H \quad \text{for complex flavors},
\]

where \( k = \min(m, n) \).

Each \( H(i) \) has the form

\[
H(i) = I - \tau_v v^v',
\]

where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(n-k+i+1:n) = 0 \) and \( v(n-k+i) = 1; v(1:n-k+i-1) \) for real flavors or \( \text{conjg}(v(1:n-k+i-1)) \) for complex flavors is stored on exit in \( A(ia+m-k+i-1, ja:ja+n-k+i-2) \), and \( \tau \) in \( \text{tau}[ia+m-k+i-2] \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?getf2**

Computes an LU factorization of a general matrix, using partial pivoting with row interchanges (local blocked algorithm).

**Syntax**

```c
void psgetf2 (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pdgetf2 (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pcgetf2 (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
void pzgetf2 (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *info);
```
Include Files

- mkl_scalapack.h

Description

The \( p\text{-}getf2 \) function computes an LU factorization of a general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) using partial pivoting with row interchanges.

The factorization has the form \( \text{sub}(A) = P \times L \times U \), where \( P \) is a permutation matrix, \( L \) is lower triangular with unit diagonal elements (lower trapezoidal if \( m > n \)), and \( U \) is upper triangular (upper trapezoidal if \( m < n \)). This is the right-looking Parallel Level 2 BLAS version of the algorithm.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\( m \) (global)
The number of rows in the distributed matrix \( \text{sub}(A) \). \( (m \geq 0) \).

\( n \) (global) The number of columns in the distributed matrix \( \text{sub}(A) \). \( (nb_a - \text{mod}(ja-1, nb_a) \geq n \geq 0) \).

\( a \) (local).

Pointer into the local memory to an array of size \( lld_a \times \text{LOC}(ja+n-1) \).

On entry, this array contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \).

\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( desca \) (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

Output Parameters

\( ipiv \) (local)

Array of size \( \text{LOC}(m_a) + mb_a \). This array contains the pivoting information. \( ipiv[i] \rightarrow \) The global row that local row \((i + 1)\) was swapped with, \( i = 0, 1, \ldots, \text{LOC}(m_a) + mb_a - 1 \). This array is tied to the distributed matrix \( A \).

\( info \) (local).

If \( info = 0 \): successful exit.
If \( \text{info} < 0 \):

- if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( \text{info} = -(i*100+j) \),
- if the \( i \)-th argument is a scalar and had an illegal value, then \( \text{info} = -i \).

If \( \text{info} > 0 \): If \( \text{info} = k \), the matrix element \( U(ia+k-1, ja+k-1) \) is exactly zero. The factorization has been completed, but the factor \( U \) is exactly singular, and division by zero will occur if it is used to solve a system of equations.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?labrd**

Reduces the first \( nb \) rows and columns of a general rectangular matrix \( A \) to real bidiagonal form by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of \( A \).

**Syntax**

```c
void pslabrd (MKL_INT *m, MKL_INT *n, MKL_INT *nb, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, float *tauq, float *taup, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, float *work);
void pdlabrd (MKL_INT *m, MKL_INT *n, MKL_INT *nb, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, double *tauq, double *taup, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, double *work);
void pclabrd (MKL_INT *m, MKL_INT *n, MKL_INT *nb, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, MKL_Complex8 *tauq, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_Complex8 *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, MKL_Complex8 *work);
void pzlabrd (MKL_INT *m, MKL_INT *n, MKL_INT *nb, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, MKL_Complex16 *tauq, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_Complex16 *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?labrd function reduces the first \( nb \) rows and columns of a real/complex general \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \) to upper or lower bidiagonal form by an orthogonal/unitary transformation \( Q^* A * P \), and returns the matrices \( X \) and \( Y \) necessary to apply the transformation to the unreduced part of \( \text{sub}(A) \).

If \( m \geq n \), \( \text{sub}(A) \) is reduced to upper bidiagonal form; if \( m < n \), \( \text{sub}(A) \) is reduced to lower bidiagonal form.

This is an auxiliary function called by p?gebrd.
Input Parameters

\( m \)  
(global) The number of rows in the distributed matrix \( \text{sub}(A) \). \((m \geq 0)\).

\( n \)  
(global) The number of columns in the distributed matrix \( \text{sub}(A) \). \((n \geq 0)\).

\( nb \)  
(global)  
The number of leading rows and columns of \( \text{sub}(A) \) to be reduced.

\( a \)  
(local).  
Pointer into the local memory to an array of size \( lld_a \times \text{LOC}(ja+n-1) \).  
On entry, this array contains the local pieces of the general distributed matrix \( \text{sub}(A) \).

\( ia, ja \)  
(global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( desca \)  
(global and local) array of size \( dlen_\) . The array descriptor for the distributed matrix \( A \).

\( ix, jx \)  
(global) The row and column indices in the global matrix \( X \) indicating the first row and the first column of the matrix \( \text{sub}(X) \), respectively.

\( descx \)  
(global and local) array of size \( dlen_\) . The array descriptor for the distributed matrix \( X \).

\( iy, jy \)  
(global) The row and column indices in the global matrix \( Y \) indicating the first row and the first column of the matrix \( \text{sub}(Y) \), respectively.

\( descy \)  
(global and local) array of size \( dlen_\) . The array descriptor for the distributed matrix \( Y \).

\( work \)  
(local).  
Workspace array of size \( lwork \).  
\( lwork \geq nb_a + nq \),  
with \( nq = \text{numroc}(n+\text{mod}(ia-1, nb_y), nb_y, mycol, iacol, npcol) \)  
\( iacol = \text{indxg2p} (ja, nb_a, mycol, csrc_a, npcol) \)  
\( \text{indxg2p} \) and \( \text{numroc} \) are \text{ScALAPACK} tool functions; \( myrow, mycol, nprow, \) and \( npcol \) can be determined by calling the function \text{blacs_gridinfo}.

Output Parameters

\( a \)  
(local)  
On exit, the first \( nb \) rows and columns of the matrix are overwritten; the rest of the distributed matrix \( \text{sub}(A) \) is unchanged.

If \( m \geq n \), elements on and below the diagonal in the first \( nb \) columns, with the array \( \text{tauq} \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors; and elements above the diagonal in the first \( nb \) rows, with the array \( \text{taup} \), represent the orthogonal/unitary matrix \( P \) as a product of elementary reflectors.
If \( m < n \), elements below the diagonal in the first \( nb \) columns, with the array \( \text{tauq} \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors, and elements on and above the diagonal in the first \( nb \) rows, with the array \( \text{taup} \), represent the orthogonal/unitary matrix \( P \) as a product of elementary reflectors. See Application Notes below.

\( d \) (local).

Array of size \( \text{LOCr}(ia+\min(m,n)-1) \) if \( m \geq n \); \( \text{LOCc}(ja+\min(m,n)-1) \) otherwise. The distributed diagonal elements of the bidiagonal distributed matrix \( B \):

\[ d[i] = A(ia+i, ja+i), \quad i = 0, 1, \ldots, \text{size}(d)-1 \]

\( d \) is tied to the distributed matrix \( A \).

\( e \) (local).

Array of size \( \text{LOCr}(ia+\min(m,n)-1) \) if \( m \geq n \); \( \text{LOCc}(ja+\min(m,n)-2) \) otherwise. The distributed off-diagonal elements of the bidiagonal distributed matrix \( B \):

if \( m \geq n \), \( e[i] = A(ia+i, ja+i+1) \) for \( i = 0, 1, \ldots, n-2 \); 
if \( m < n \), \( e[i] = A(ia+i+1, ja+i) \) for \( i = 0, 1, \ldots, m-2 \).

\( e \) is tied to the distributed matrix \( A \).

\( \text{tauq}, \text{taup} \) (local).

Array size \( \text{LOCc}(ja+\min(m,n)-1) \) for \( \text{tauq} \), size \( \text{LOCr}(ia+\min(m,n)-1) \) for \( \text{taup} \). The scalar factors of the elementary reflectors which represent the orthogonal/unitary matrix \( Q \) for \( \text{tauq} \), \( P \) for \( \text{taup} \). \( \text{tauq} \) and \( \text{taup} \) are tied to the distributed matrix \( A \). See Application Notes below.

\( x \) (local)

Pointer into the local memory to an array of size \( lld_x \times nb \). On exit, the local pieces of the distributed \( m \)-by-\( nb \) matrix \( X(ix:ix+m-1, jx:jx+nb-1) \) required to update the unreduced part of \( \text{sub}(A) \).

\( y \) (local)

Pointer into the local memory to an array of size \( lld_y \times nb \). On exit, the local pieces of the distributed \( n \)-by-\( nb \) matrix \( Y(iy:iy+n-1, jy:jy+nb-1) \) required to update the unreduced part of \( \text{sub}(A) \).

Application Notes

The matrices \( Q \) and \( P \) are represented as products of elementary reflectors:

\[
Q = H(1)*H(2)*\ldots*H(nb), \quad \text{and} \quad P = G(1)*G(2)*\ldots*G(nb)
\]

Each \( H(i) \) and \( G(i) \) has the form:

\[
H(i) = I - \text{tauq}^*v*v', \quad \text{and} \quad G(i) = I - \text{taup}^*u*u',
\]

where \( \text{tauq} \) and \( \text{taup} \) are real/complex scalars, and \( v \) and \( u \) are real/complex vectors.

If \( m \geq n \), \( v(1: i-1) = 0 \), \( v(i) = 1 \), and \( v(i:m) \) is stored on exit in
\[
A(ia+i-1:ia+m-1, ja+i-1); \quad u(1:i) = 0, \quad u(i+1) = 1, \quad \text{and} \quad u(i+1:n) \text{ is stored on exit in } A(ia+i-1, ja+i:ja+n-1); \quad \text{tauq is stored in } \text{tauq}[ja+i-2] \quad \text{and} \quad \text{taup in } \text{taup}[ia+i-2].
\]
If $m < n$, $v(1:i) = 0, v(i+1) = 1$, and $v(i+1:m)$ is stored on exit in $A(ia+i:ia+m-1, ja+i-1); u(1:i-1) = 0, u(i) = 1$, and $u(i:n)$ is stored on exit in $A(ia+i-1, ja+i:j+a+n-1); taq$ is stored in $taq(ia+i-2)$ and $taup$ in $taup(ia+i-2)$. The elements of the vectors $v$ and $u$ together form the $m$-by-$nb$ matrix $V$ and the $nb$-by-$n$ matrix $U'$ which are necessary, with $X$ and $Y$, to apply the transformation to the unreduced part of the matrix, using a block update of the form: $sub(A) := sub(A) - V*Y' - X*U'$. The contents of $sub(A)$ on exit are illustrated by the following examples with $nb = 2$:

$$
\begin{matrix}
m = 6 \mod n = 6 \geq nb & m = 5 \mod n = 5 \geq nb \\
1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 \\
v1 & 1 & 2 & 1 & 2 & 1 \\
v1 & 2 & 1 & 2 & 1 & 2 \\
v1 & 2 & 1 & 2 & 1 & 2 \\
v1 & 2 & 1 & 2 & 1 & 2 \\
\end{matrix}
$$

where $a$ denotes an element of the original matrix which is unchanged, $vi$ denotes an element of the vector defining $H(i)$, and $ui$ an element of the vector defining $G(i)$.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacon
Estimates the 1-norm of a square matrix, using the reverse communication for evaluating matrix-vector products.

Syntax

void pslacon (MKL_INT *n, float *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *isgn, float *est, MKL_INT *kase);

void pdlacon (MKL_INT *n, double *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *isgn, double *est, MKL_INT *kase);

void pclacon (MKL_INT *n, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *est, MKL_INT *kase);

void pzlacon (MKL_INT *n, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *est, MKL_INT *kase);

Include Files
• mkl_scalapack.h

Description
The p?lacon function estimates the 1-norm of a square, real/unitary distributed matrix A. Reverse communication is used for evaluating matrix-vector products. $x$ and $v$ are aligned with the distributed matrix $A$, this information is implicitly contained within $iv, ix, descv$, and $descx$.

Input Parameters

$n$

(global) The length of the distributed vectors $v$ and $x. n \geq 0.$
v (local).
Pointer into the local memory to an array of size $\text{LOCr}(n + \text{mod}(iv-1, \text{mb}_v))$.
On the final return, $v = a^*w$, where $est = \text{norm}(v)/\text{norm}(w)$ ($w$ is not returned).

$iv, jv$ (global) The row and column indices in the global matrix $V$ indicating the first row and the first column of the submatrix $V$, respectively.

descv (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $V$.

$x$ (local).
Pointer into the local memory to an array of size $\text{LOCr}(n + \text{mod}(ix-1, \text{mb}_x))$.

$ix, jx$ (global) The row and column indices in the global matrix $X$ indicating the first row and the first column of the submatrix $X$, respectively.

descx (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $X$.

$\text{isgn}$ (local).
Array of size $\text{LOCr}(n + \text{mod}(ix-1, \text{mb}_x))$. $\text{isgn}$ is aligned with $x$ and $v$.

$kase$ (local).
On the initial call to $\text{p?lacon}$, $kase$ should be 0.

**Output Parameters**

$x$ (local).
On an intermediate return, $X$ should be overwritten by $A^*X$, if $kase=1$, $A^T*X$, if $kase=2$,
$p?\text{lacon}$ must be re-called with all the other parameters unchanged.

$est$ (global).

$kase$ (local)
On an intermediate return, $kase$ is 1 or 2, indicating whether $X$ should be overwritten by $A^*X$, or $A^{\prime}\_\_\_\_\_\_X$. On the final return from $p?\text{lacon}$, $kase$ is again 0.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?laconsb**

*Looks for two consecutive small subdiagonal elements.*

**Syntax**

```c
void pslaconsb (const float *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l,
                const float *m, const float *h44, const float *h33, const float *h43h34, float *buf,
                const MKL_INT *lwork );
```
void pdlaconsb (const double *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *m, const double *h44, const double *h33, const double *h43h34, double *buf, const MKL_INT *lwork);

void pclaconsb (const MKL_Complex8 *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *m, const MKL_Complex8 *h44, const MKL_Complex8 *h33, const MKL_Complex8 *h43h34, MKL_Complex8 *buf, const MKL_INT *lwork);

void pzlaconsb (const MKL_Complex16 *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *m, const MKL_Complex16 *h44, const MKL_Complex16 *h33, const MKL_Complex16 *h43h34, MKL_Complex16 *buf, const MKL_INT *lwork);

Include Files
• mkl_scalapack.h

Description
The p?laconsb function looks for two consecutive small subdiagonal elements by analyzing the effect of starting a double shift QR iteration given by \( h_{44}, h_{33}, \) and \( h_{43}h_{34} \) to see if this process makes a subdiagonal negligible.

Input Parameters

\( a \) (local)
Array of size \( lld_a*LOCc(n_a) \). On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

\( \text{desca} \) (global and local)
Array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( i \) (global)
The global location of the bottom of the unreduced submatrix of \( A \). Unchanged on exit.

\( l \) (global)
The global location of the top of the unreduced submatrix of \( A \). Unchanged on exit.

\( h_{44}, h_{33}, h_{43}h_{34} \) (global).
These three values are for the double shift QR iteration.

\( lwork \) (local)
This must be at least \( 7*\text{ceil}(\text{ceil}(i-l)/\text{mb_a})/\text{lcm} (\text{nprow}, \text{npcol}) \). Here \( \text{lcm} \) is the least common multiple and \( \text{nprow}*\text{npcol} \) is the logical grid size.

Output Parameters

\( m \) (global). On exit, this yields the starting location of the QR double shift.
This will satisfy:

\[ l \leq m \leq i-2. \]

\( \text{buf} \) (local).
Array of size \( lwork \).
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?lacp2}

\textit{Copies all or part of a distributed matrix to another distributed matrix.}

\textbf{Syntax}

\begin{verbatim}
void pslacp2 (char *uplo, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb);
void pdlacp2 (char *uplo, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb);
void pclacp2 (char *uplo, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb);
void pzlacp2 (char *uplo, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb);
\end{verbatim}

\textit{Include Files}

- mkl_scalapack.h

\textbf{Description}

The \texttt{p?lacp2} function copies all or part of a distributed matrix $A$ to another distributed matrix $B$. No communication is performed, \texttt{p?lacp2} performs a local copy sub($A$):= sub($B$), where sub($A$) denotes $A(ia:ia+m-1,a:ja+n-1)$ and sub($B$) denotes $B(ib:ib+m-1,jb:jb+n-1)$.

\texttt{p?lacp2} requires that only dimension of the matrix operands is distributed.

\textbf{Input Parameters}

\begin{itemize}
  \item \texttt{uplo} (global) Specifies the part of the distributed matrix sub($A$) to be copied:
    \begin{itemize}
      \item = \text{'U'}: Upper triangular part is copied; the strictly lower triangular part of sub($A$) is not referenced;
      \item = \text{'L'}: Lower triangular part is copied; the strictly upper triangular part of sub($A$) is not referenced.
    \end{itemize}
  \item \texttt{m} (global)
The number of rows in the distributed matrix sub($A$). ($m \geq 0$).
  \item \texttt{n} (global)
The number of columns in the distributed matrix sub($A$). ($n \geq 0$).
  \item \texttt{a} (local).
Point into the local memory to an array of size $\text{lld}_a * \text{LOCc}(ja+n-1)$.
\end{itemize}

On entry, this array contains the local pieces of the $m$-by-$n$ distributed matrix sub($A$).
**ia, ja**
(global) The row and column indices in the global matrix A indicating the first row and the first column of sub(A), respectively.

**desca**
(global and local) array of size dlen_. The array descriptor for the distributed matrix A.

**ib, jb**
(global) The row and column indices in the global matrix B indicating the first row and the first column of sub(B), respectively.

**descb**
(global and local) array of size dlen_. The array descriptor for the distributed matrix B.

**Output Parameters**

**b**
(local).

Pointer into the local memory to an array of size lld_b * LOCc(jb+n-1). This array contains on exit the local pieces of the distributed matrix sub(B) set as follows:

- if uplo = 'U', \(B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), 1 \leq i \leq j, 1 \leq j \leq n;\)
- if uplo = 'L', \(B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), j \leq i \leq m, 1 \leq j \leq n;\)
- otherwise, \(B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1), 1 \leq i \leq m, 1 \leq j \leq n.\)

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?lacp3**

Copies from a global parallel array into a local replicated array or vice versa.

**Syntax**

```c
void pslacp3 (const MKL_INT *m, const MKL_INT *i, float *a, const MKL_INT *desca, float *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *rev);
void pdlacp3 (const MKL_INT *m, const MKL_INT *i, double *a, const MKL_INT *desca, double *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *rev);
void pclacp3 (const MKL_INT *m, const MKL_INT *i, MKL_Complex8 *a, const MKL_INT *desca, MKL_Complex8 *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *rev);
void pzlacp3 (const MKL_INT *m, const MKL_INT *i, MKL_Complex16 *a, const MKL_INT *desca, MKL_Complex16 *b, const MKL_INT *ldb, const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *rev);
```

**Include Files**

- mkl_scalapack.h

**Description**

This is an auxiliary function that copies from a global parallel array into a local replicated array or vice versa. Note that the entire submatrix that is copied gets placed on one node or more. The receiving node can be specified precisely, or all nodes can receive, or just one row or column of nodes.
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

$m$ (global)

$m$ is the order of the square submatrix that is copied.

$m \geq 0$. Unchanged on exit.

$i$ (global)
The matrix element $A(i, i)$ is the global location that the copying starts from. Unchanged on exit.

$a$ (local)

Array of size $lld_a \times LOCc(n_a)$. On entry, the parallel matrix to be copied into or from.

$desca$ (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$b$ (local)

Array of size $ldb \times LOCc(m)$. If $rev = 0$, this is the global portion of the matrix $A(i:i+m-1, i:i+m-1)$. If $rev = 1$, this is unchanged on exit.

$ldb$ (local)
The leading dimension of $B$.

$ii, jj$ (global) By using $rev$ 0 and 1, data can be sent out and returned again. If $rev = 0$, then $ii$ is destination row index and $jj$ is destination column index for the node(s) receiving the replicated matrix$B$. If $ii \geq 0$, $jj \geq 0$, then node $(ii, jj)$ receives the data. If $ii = -1$, $jj \geq 0$, then all rows in column $jj$ receive the data. If $ii \geq 0$, $jj = -1$, then all cols in row $ii$ receive the data. If $ii = -1$, $jj = -1$, then all nodes receive the data. If $rev != 0$, then $ii$ is the source row index for the node(s) sending the replicated $B$.

$rev$ (global) Use $rev = 0$ to send global matrix$A$ into locally replicated matrix$B$ (on node $(ii, jj)$). Use $rev != 0$ to send locally replicated $B$ from node $(ii, jj)$ to its owner (which changes depending on its location in $A$) into the global $A$.

Output Parameters

$a$

On exit, if $rev = 1$, the copied data. Unchanged on exit if $rev = 0$.

$b$

If $rev = 1$, this is unchanged on exit.
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lacpy
Copies all or part of one two-dimensional array to another.

Syntax
void pslacpy (char *uplo , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pdlacpy (char *uplo , MKL_INT *m , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pclacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );
void pzlacpy (char *uplo , MKL_INT *m , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb );

Include Files
• mkl_scalapack.h

Description
The p?lacpy function copies all or part of a distributed matrix \( A \) to another distributed matrix \( B \). No communication is performed, \( \texttt{p?lacpy} \) performs a local copy sub(\( B \)) := sub(\( A \)), where sub(\( A \)) denotes \( A(ia:ia+m-1, ja:ja+n-1) \) and sub(\( B \)) denotes \( B(ib:ib+m-1, jb:jb+n-1) \).

Input Parameters

uplo
(global) Specifies the part of the distributed matrix sub(\( A \)) to be copied:
\begin{itemize}
  \item = 'U': Upper triangular part; the strictly lower triangular part of sub(\( A \)) is not referenced;
  \item = 'L': Lower triangular part; the strictly upper triangular part of sub(\( A \)) is not referenced.
\end{itemize}
Otherwise: all of the matrix sub(\( A \)) is copied.

m
(global)
The number of rows in the distributed matrix sub(\( A \)). \((m\geq0)\).

n
(global)
The number of columns in the distributed matrix sub(\( A \)). \((n\geq0)\).

a
(local).
Pointer into the local memory to an array of size \( lld_a * LOCc(ja+n-1) \).

On entry, this array contains the local pieces of the distributed matrix sub(\( A \)).
ia, ja  (global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the matrix $\text{sub}(A)$, respectively.

desca  (global and local) array of size $\text{dlen}_a$. The array descriptor for the distributed matrix $A$.

ib, jb  (global) The row and column indices in the global matrix $B$ indicating the first row and the first column of $\text{sub}(B)$ respectively.

descb  (global and local) array of size $\text{dlen}_b$. The array descriptor for the distributed matrix $B$.

Output Parameters

b  (local).

 Pointer into the local memory to an array of size $\text{lld}_b * \text{LOCc}(\text{jb}+\text{n}-1)$. This array contains on exit the local pieces of the distributed matrix $\text{sub}(B)$ set as follows:

if $\text{uplo} = 'U'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq j$, $1 \leq j \leq n$;

if $\text{uplo} = 'L'$, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $j \leq i \leq m$, $1 \leq j \leq n$;

otherwise, $B(ib+i-1, jb+j-1) = A(ia+i-1, ja+j-1)$, $1 \leq i \leq m$, $1 \leq j \leq n$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laevswp

Moves the eigenvectors from where they are computed to ScaLAPACK standard block cyclic array.

Syntax

void pslaevswp (MKL_INT *n, float *zin, MKL_INT *ldzi, float *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_INT *nvs, MKL_INT *key, float *work, MKL_INT *lwork);

void pdlaevswp (MKL_INT *n, double *zin, MKL_INT *ldzi, double *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_INT *nvs, MKL_INT *key, double *work, MKL_INT *lwork);

void pclaevswp (MKL_INT *n, float *zin, MKL_INT *ldzi, MKL_Complex8 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_INT *nvs, MKL_INT *key, float *rwork, MKL_INT *lrwork);

void pzlaevswp (MKL_INT *n, double *zin, MKL_INT *ldzi, MKL_Complex16 *z, MKL_INT *iz, MKL_INT *jz, MKL_INT *descz, MKL_INT *nvs, MKL_INT *key, double *rwork, MKL_INT *lrwork);

Include Files

• mkl_scalapack.h

Description

The p?laevswp function moves the eigenvectors (potentially unsorted) from where they are computed, to a ScaLAPACK standard block cyclic array, sorted so that the corresponding eigenvalues are sorted.
**Input Parameters**

\( np \) = the number of rows local to a given process.

\( nq \) = the number of columns local to a given process.

\( n \)  

The order of the matrix \( A \). \( n \geq 0 \).

\( zin \)  

Array of size \( ldzi \times nvs[iam+1] \). The eigenvectors on input. \( iam \) is a process rank from \([0, nprocs)\) interval. Each eigenvector resides entirely in one process. Each process holds a contiguous set of \( nvs[iam+1] \) eigenvectors. The global number of the first eigenvector that the process holds is: \((\text{sum for } i=[0, iam] \text{ of } nvs[i]) + 1\).

\( ldzi \)  

The leading dimension of the \( zin \) array.

\( iz, jz \)  

The row and column indices in the global matrix \( Z \) indicating the first row and the first column of the submatrix \( Z \), respectively.

\( descz \)  

Array of size \( dlen_\cdot \). The array descriptor for the distributed matrix \( Z \).

\( nvs \)  

Array of size \( nprocs+1 \)

\( nvs[i] = \text{number of eigenvectors held by processes } [0, i) \)

\( nvs[0] = \text{number of eigenvectors held by processes } [0, 0) = 0 \)

\( nvs[nprocs] = \text{number of eigenvectors held by } [0, nprocs) = \text{total number of eigenvectors.} \)

\( key \)  

Array of size \( n \). Indicates the actual index (after sorting) for each of the eigenvectors.

\( rwork \)  

Array of size \( lrwork \).

\( lrwork \)  

Size of \( work \).

**Output Parameters**

\( z \)  

Array of global size \( n \times n \) and of local size \( lld_z \times nq \). The eigenvectors on output. The eigenvectors are distributed in a block cyclic manner in both dimensions, with a block size of \( nb \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?lahrd**

Reduces the first \( nb \) columns of a general rectangular matrix \( A \) so that elements below the \( k \)-th subdiagonal are zero, by an orthogonal/unitary transformation, and returns auxiliary matrices that are needed to apply the transformation to the unreduced part of \( A \).

**Syntax**

```c
void pslahrd (MKL_INT *n, MKL_INT *k, MKL_INT *nb, float *a, MKL_INT *ia, MKL_INT *desca, float *tau, float *t, float *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, float *work);

void pdlahrd (MKL_INT *n, MKL_INT *k, MKL_INT *nb, double *a, MKL_INT *ia, MKL_INT *desca, double *tau, double *t, double *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, double *work);

void pclahrd (MKL_INT *n, MKL_INT *k, MKL_INT *nb, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *t, MKL_Complex8 *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, MKL_Complex8 *work);

void pzlahrd (MKL_INT *n, MKL_INT *k, MKL_INT *nb, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *t, MKL_Complex16 *y, MKL_INT *iy, MKL_INT *jy, MKL_INT *descy, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The \( p?lahrd \) function reduces the first \( nb \) columns of a real general \( n \)-by-\((n-k+1) \) distributed matrix \( A(ia:ia+n-1, ja:ja+n-k) \) so that elements below the \( k \)-th subdiagonal are zero. The reduction is performed by an orthogonal/unitary similarity transformation \( Q^*A*Q \). The function returns the matrices \( V \) and \( T \) which determine \( Q \) as a block reflector \( I-V*T*V^* \), and also the matrix \( Y = A*V*T \).

This is an auxiliary function called by \( p?gehrd \). In the following comments \( \text{sub}(A) \) denotes \( A(ia:ia+n-1, ja:ja+n-1) \).

**Input Parameters**

- \( n \)
  
  (global)
  
  The order of the distributed matrix \( \text{sub}(A) \). \( n \geq 0 \).

- \( k \)
  
  (global)
  
  The offset for the reduction. Elements below the \( k \)-th subdiagonal in the first \( nb \) columns are reduced to zero.

- \( nb \)
  
  (global)
  
  The number of columns to be reduced.

- \( a \)
  
  (local)
  
  Pointer into the local memory to an array of size \( lld_a \ast LOCC(ja+n-k) \). On entry, this array contains the local pieces of the \( n \)-by-\((n-k+1) \) general distributed matrix \( A(ia:ia+n-1, ja:ja+n-k) \).
\( ia, ja \) (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\cdot \). The array descriptor for the distributed matrix \( A \).

\( iy, jy \) (global) The row and column indices in the global matrix \( Y \) indicating the first row and the first column of the matrix \( \text{sub}(Y) \), respectively.

\( \text{descy} \) (global and local) array of size \( dlen_\cdot \). The array descriptor for the distributed matrix \( Y \).

\( \text{work} \) (local). Array of size \( nb \).

**Output Parameters**

\( a \) (local).

On exit, the elements on and above the \( k \)-th subdiagonal in the first \( nb \) columns are overwritten with the corresponding elements of the reduced distributed matrix; the elements below the \( k \)-th subdiagonal, with the array \( \tau \), represent the matrix \( Q \) as a product of elementary reflectors. The other columns of the matrix \( A(ia:ia+n-1, ja:ja+n-k) \) are unchanged. (See **Application Notes** below.)

\( \tau \) (local)

Array of size \( \text{LOCc}(ja+n-2) \). The scalar factors of the elementary reflectors (see **Application Notes** below). \( \tau \) is tied to the distributed matrix \( A \).

\( t \) (local)

Array of size \( nb_a * nb_a \). The upper triangular matrix \( T \).

\( y \) (local).

Pointer into the local memory to an array of size \( lld_y * nb_a \). On exit, this array contains the local pieces of the \( n \)-by-\( nb \) distributed matrix \( Y \). \( lld_y \geq \text{LOCr}(ia+n-1) \).

**Application Notes**

The matrix \( Q \) is represented as a product of \( nb \) elementary reflectors

\[ Q = H(1)^*H(2)^*...*H(nb)^*. \]

Each \( H(i) \) has the form

\[ H(i) = i-\tau*v*v^\prime, \]

where \( \tau \) is a real/complex scalar, and \( v \) is a real/complex vector with \( v(1:i+k-1)=0, v(i+k)=1; v(i+k+1:n) \) is stored on exit in \( A(a+i+k:ia+n-1, ja+i-1) \), and \( \tau \) in \( \tau[ja+i-2] \).

The elements of the vectors \( v \) together form the \( (n-k+1) \)-by-\( nb \) matrix \( V \) which is needed, with \( T \) and \( Y \), to apply the transformation to the unreduced part of the matrix, using an update of the form: \( A(ia:ia+n-1, ja:ja+n-k) := (I-V^*T^*V)*(A(ia:ia+n-1, ja:ja+n-k)-Y^*V) \). The contents of \( A(ia:ia+n-1, ja:ja+n-k) \) on exit are illustrated by the following example with \( n = 7, k = 3 \), and \( nb = 2 \):
where $a$ denotes an element of the original matrix $A(ia:ia+n-1, ja:ja+n-k)$, $h$ denotes a modified element of the upper Hessenberg matrix $H$, and $v_i$ denotes an element of the vector defining $H(i)$.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

### p?laiect

*Exploits IEEE arithmetic to accelerate the computations of eigenvalues.*

#### Syntax

```c
void pslaiect (float *sigma, MKL_INT *n, float *d, MKL_INT *count);
void pdlaiectb (float *sigma, MKL_INT *n, float *d, MKL_INT *count);
void pdlaiectl (float *sigma, MKL_INT *n, float *d, MKL_INT *count);
```

#### Include Files

- mkl_scalapack.h

#### Description

The p?laiect function computes the number of negative eigenvalues of $(A-\sigma I)$. This implementation of the Sturm Sequence loop exploits IEEE arithmetic and has no conditionals in the innermost loop. The signbit for real function pslaiect is assumed to be bit 32. Double-precision functions pdlaiectb and pdlaiectl differ in the order of the double precision word storage and, consequently, in the signbit location. For pdlaiectb, the double precision word is stored in the big-endian word order and the signbit is assumed to be bit 32. For pdlaiectl, the double precision word is stored in the little-endian word order and the signbit is assumed to be bit 64.

This is a ScaLAPACK internal function and arguments are not checked for unreasonable values.

#### Input Parameters

- **sigma**
  
  The shift. p?laiect finds the number of eigenvalues less than equal to $\sigma$.

- **n**
  
  The order of the tridiagonal matrix $T$. $n \geq 1$.

- **d**
  
  Array of size $2n-1$.

  On entry, this array contains the diagonals and the squares of the off-diagonal elements of the tridiagonal matrix $T$. These elements are assumed to be interleaved in memory for better cache performance. The diagonal entries of $T$ are in the entries $d[0]$, $d[2]$, ..., $d[2n-2]$, while the squares of the off-diagonal entries are $d[1]$, $d[3]$, ..., $d[2n-3]$. To
avoid overflow, the matrix must be scaled so that its largest entry is no
greater than $\text{overflow}^{(1/2)} \ast \text{underflow}^{(1/4)}$ in absolute value, and for
greatest accuracy, it should not be much smaller than that.

**Output Parameters**

\(n\)  
The count of the number of eigenvalues of \(T\) less than or equal to \(sigma\).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

### p?lamve

*Copies all or part of one two-dimensional distributed array to another.*

#### Syntax

```c
void pslamve(char* uplo, MKL_INT* m, MKL_INT* n, float* a, MKL_INT* ia, MKL_INT* ja,
MKL_INT* desca, float* b, MKL_INT* ib, MKL_INT* jb, MKL_INT* descb, float* dwork);
void pdlamve(char* uplo, MKL_INT* m, MKL_INT* n, double* a, MKL_INT* ia, MKL_INT* ja,
MKL_INT* desca, double* b, MKL_INT* ib, MKL_INT* jb, MKL_INT* descb, double* dwork);
```

#### Include Files

- mkl_scalapack.h

**Description**

\(p?lamve\) copies all or part of a distributed matrix \(A\) to another distributed matrix \(B\). There is no alignment assumptions at all except that \(A\) and \(B\) are of the same size.

### Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

\(uplo\)  
*(global )*

Specifies the part of the distributed matrix \(\text{sub}(A)\) to be copied:

- If \(= 'U'\): Upper triangular part is copied; the strictly lower triangular part of \(\text{sub}(A)\) is not referenced;
- If \(= 'L'\): Lower triangular part is copied; the strictly upper triangular part of \(\text{sub}(A)\) is not referenced;

Otherwise: All of the matrix \(\text{sub}(A)\) is copied.
The number of rows to be operated on, which is the number of rows of the distributed matrix \( \text{sub}(A) \). \( m \geq 0 \).

The number of columns to be operated on, which is the number of columns of the distributed matrix \( \text{sub}(A) \). \( n \geq 0 \).

(\text{local} ) pointer into the local memory to an array of size \( lld_a \times LOC_c(ja + n - 1) \). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \) to be copied from.

The row index in the global matrix \( A \) indicating the first row of \( \text{sub}(A) \).

The column index in the global matrix \( A \) indicating the first column of \( \text{sub}(A) \).

(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

The row index in the global matrix \( B \) indicating the first row of \( \text{sub}(B) \).

The column index in the global matrix \( B \) indicating the first column of \( \text{sub}(B) \).

(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( B \).

(\text{local workspace} ) array

If \( \text{uplo} = 'U' \) or \( \text{uplo} = 'L' \) and number of processors > 1, the length of \( dwork \) is at least as large as the length of \( b \). Otherwise, \( dwork \) is not referenced.

(\text{local} ) pointer into the local memory to an array of size \( lld_b \times LOC_c(jb + n - 1) \). This array contains on exit the local pieces of the distributed matrix \( \text{sub}(B) \).

\textbf{See Also}

\textbf{Overview} for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?lange}

\textit{Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a general rectangular matrix.}
Syntax

```c
float pslange (char *norm, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pdlange (char *norm, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
float pclange (char *norm, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pzlange (char *norm, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
```

Include Files

- mkl_scalapack.h

Description

The `p?lange` function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix \( \text{sub}(A) = A(\text{ia:ia+m-1, ja:ja+n-1}) \).

Input Parameters

- **norm**
  (global) Specifies what value is returned by the function:
  - 'M' or 'm': \( \text{val} = \max(|A_{ij}|) \), largest absolute value of the matrix \( A \), it is not a matrix norm.
  - '1' or 'O' or 'o': \( \text{val} = \text{norm}_1(A) \), 1-norm of the matrix \( A \) (maximum column sum),
  - 'I' or 'i': \( \text{val} = \text{norm}_\infty(A) \), infinity norm of the matrix \( A \) (maximum row sum),
  - 'F', 'f', 'E' or 'e': \( \text{val} = \text{norm}_F(A) \), Frobenius norm of the matrix \( A \) (square root of sum of squares).

- **m**
  (global)
  The number of rows in the distributed matrix \( \text{sub}(A) \). When \( m = 0 \), `p?lange` is set to zero. \( m \geq 0 \).

- **n**
  (global)
  The number of columns in the distributed matrix \( \text{sub}(A) \). When \( n = 0 \), `p?lange` is set to zero. \( n \geq 0 \).

- **a**
  (local).
  Pointer into the local memory to an array of size \( \text{lld}_a \times \text{LOCc}(ja+n-1) \) containing the local pieces of the distributed matrix \( A \).

- **ia, ja**
  (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

- **desca**
  (global and local) array of size \( dlen_\_) \). The array descriptor for the distributed matrix \( A \).

- **work**
  (local).
  Array size \( lwork \).
\( l_{\text{work}} \geq 0 \) if \( \text{norm} = 'M' \) or \( 'm' \) (not referenced),
\( n_{\text{q}}0 \) if \( \text{norm} = '1', 'O' \) or \( 'o' \),
\( m_{\text{p}}0 \) if \( \text{norm} = 'I' \) or \( 'i' \),
\( 0 \) if \( \text{norm} = 'F', 'f', 'E' \) or \( 'e' \) (not referenced),

where
\[ i_{\text{roffa}} = \text{mod}(ia-1, mb_a), \quad i_{\text{coffa}} = \text{mod}(ja-1, nb_a), \]
\[ i_{\text{arow}} = \text{indxg2p}(ia, mb_a, myrow, rsrca_n, nprow), \]
\[ i_{\text{icol}} = \text{indxg2p}(ja, nb_a, mycol, csrca_n, npcol), \]
\[ m_{\text{p}}0 = \text{numroc}(m+i_{\text{roffa}}, mb_a, myrow, i_{\text{arow}}, nprow), \]
\[ n_{\text{q}}0 = \text{numroc}(n+i_{\text{coffa}}, nb_a, mycol, i_{\text{icol}}, npcol), \]

\text{indxg2p} \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; } myrow, mycol, nprow, \text{ and } npcol \text{ can be determined by calling the function } \text{blacs_gridinfo}. 

Output Parameters

\text{val} \\
\text{The value returned by the function.}

See Also

Overview \text{ for details of ScaLAPACK array descriptor structures and related notations.}

\text{p?lanhs} \\
\text{Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of an upper Hessenberg matrix.}

Syntax

\text{float pslanhs (char *norm, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);} \\
\text{double pdlanhs (char *norm, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);} \\
\text{float pclanhs (char *norm, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);} \\
\text{double pzlanhs (char *norm, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);} \\

Include Files

\text{• mkl_scalapack.h}

Description

The \text{p?lanhs} function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of an upper Hessenberg distributed matrix \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \).
Input Parameters

**norm**
Specifies the value to be returned by the function:
- `'M'` or `'m'`: \( \text{val} = \max(\text{abs}(A_{ij})) \), largest absolute value of the matrix \( A \).
- `'1'` or `'O'` or `'o'`: \( \text{val} = \text{norml}(A) \), 1-norm of the matrix \( A \) (maximum column sum),
- `'I'` or `'i'`: \( \text{val} = \text{normI}(A) \), infinity norm of the matrix \( A \) (maximum row sum),
- `'F'`, `'f'`, `'E'` or `'e'`: \( \text{val} = \text{normF}(A) \), Frobenius norm of the matrix \( A \) (square root of sum of squares).

**n**
The number of columns in the distributed matrix \( \text{sub}(A) \). When \( n = 0 \), \( p?lanhs \) is set to zero. \( n \geq 0 \).

**a**
Pointer into the local memory to an array of size \( \text{lld}_a \ast \text{LOCc}(ja+n-1) \) containing the local pieces of the distributed matrix \( \text{sub}(A) \).

**ia, ja**
The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

**desca**
(global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

**work**
(local).

Array of size \( lwork \).
- \( lwork \geq 0 \) if \( \text{norm} = 'M' \) or `'m'` (not referenced),
- \( nq0 \) if \( \text{norm} = '1', 'O' \) or `'o'`,
- \( mp0 \) if \( \text{norm} = 'I' \) or `'i'`,
- 0 if \( \text{norm} = 'F', 'f', 'E' \) or `'e'` (not referenced),

where
- \( iroffa = \text{mod}(ia-1, mb_a) \), \( icoffa = \text{mod}(ja-1, nb_a) \),
- \( iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow) \),
- \( iacol = \text{indxg2p}(ja, nb_a, mycol, csrca, npcol) \),
- \( mp0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nq0) \),
- \( nq0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol) \).

\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( myrow, imycol, nprow, \) and \( npcol \) can be determined by calling the function \( \text{blacs_gridinfo} \).

Output Parameters

**val**
The value returned by the function.
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?lansy, p?lanhe**

*Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a real symmetric or a complex Hermitian matrix.*

**Syntax**

```c
float pslansy (char *norm, char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pdlansy (char *norm, char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
float pclansy (char *norm, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pzlansy (char *norm, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
float pclanhe (char *norm, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pzlanhe (char *norm, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?lansy` and `p?lanhe` functions return the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a distributed matrix `sub(A) = A(ia:ia+m-1, ja:ja +n-1)`.

**Input Parameters**

- **norm**
  
  (global) Specifies what value is returned by the function:
  
  - '=M' or 'm': `val = max(abs(A_{ij}))`, largest absolute value of the matrix `A`, it is not a matrix norm.
  - '=1' or 'O' or 'o': `val = norm1(A)`, 1-norm of the matrix `A` (maximum column sum),
  - '=I' or 'i': `val = normI(A)`, infinity norm of the matrix `A` (maximum row sum),
  - '=F', 'f', 'E' or 'e': `val = normF(A)`, Frobenius norm of the matrix `A` (square root of sum of squares).

- **uplo**
  
  (global) Specifies whether the upper or lower triangular part of the symmetric matrix `sub(A)` is to be referenced.
  
  - '=U': Upper triangular part of `sub(A)` is referenced,
  - '=L': Lower triangular part of `sub(A)` is referenced.
The number of columns in the distributed matrix \( \text{sub}(A) \). When \( n = 0 \), \( \text{p?lansy} \) is set to zero. \( n \geq 0 \).

\( \text{a} \)

Pointer into the local memory to an array of size \( \text{lld}_a \times \text{LOCc}(ja+n-1) \) containing the local pieces of the distributed matrix \( \text{sub}(A) \).

If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular matrix whose norm is to be computed, and the strictly lower triangular part of this matrix is not referenced. If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( \text{sub}(A) \) contains the lower triangular matrix whose norm is to be computed, and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

\( \text{ia, ja} \)

The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( \text{desca} \)

Array descriptor for the distributed matrix \( A \).

\( \text{work} \)

Array of size \( lwork \).

\( lwork \geq 0 \) if \( \text{norm} = 'M' \) or \( 'm' \) (not referenced),

\( 2 \times nq0 + mp0 + ldw \) if \( \text{norm} = '1', 'O', 'o', 'I', 'i', 'E', 'e' \),

where \( ldw \) is given by:

\[
\begin{align*}
\text{if} ( \text{nprov} \neq \text{npcol} ) \text{ then} \\
\text{ldw} &= \text{mb}_a \times \text{ceil}(\text{ceil}(np0, mb_a), (\text{lcm}/\text{nprov})) \\
\text{else} \\
\text{ldw} &= 0
\end{align*}
\]

0 if \( \text{norm} = 'F', 'f', 'E', 'e' \) (not referenced),

where \( \text{lcm} \) is the least common multiple of \( \text{nprov} \) and \( \text{npcol} \), \( \text{lcm} = \text{iilcm}(\text{nprov, npcol}) \) and \( \text{ceil}(x, y) \) is a \( \text{Scalapack} \) function that returns ceiling \( (x/y) \).

\( \text{iroffa} = \text{mod}(\text{ia} - 1, \text{mb}_a) \), \( \text{icoffa} = \text{mod}(\text{ja} - 1, \text{nb}_a) \),

\( \text{iarow} = \text{indxg2p}(\text{ia}, \text{mb}_a, \text{myrow}, \text{rsrc}_a, \text{nprow}) \),

\( \text{iacol} = \text{indxg2p}(\text{ja}, \text{nb}_a, \text{mycol}, \text{csrc}_a, \text{npcol}) \),

\( \text{mp0} = \text{numroc}(\text{m} + \text{iroffa}, \text{mb}_a, \text{myrow}, \text{iarow}, \text{nprow}) \),

\( \text{nq0} = \text{numroc}(\text{n} + \text{icoffa}, \text{nb}_a, \text{mycol}, \text{iacol}, \text{npcol}) \),

\( \text{iilcm}, \text{iceil}, \text{indxg2p}, \text{and} \text{numroc} \) are \( \text{Scalapack} \) tool functions; \( \text{myrow}, \text{mycol}, \text{nprow}, \text{and} \text{npcol} \) can be determined by calling the function \( \text{blacs_gridinfo} \).
Output Parameters

val  
The value returned by the function.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lantr
Returns the value of the 1-norm, Frobenius norm, infinity-norm, or the largest absolute value of any element, of a triangular matrix.

Syntax

float pslantr (char *norm, char *uplo, char *diag, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pdlantr (char *norm, char *uplo, char *diag, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);
float pclantr (char *norm, char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *work);
double pzlantr (char *norm, char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *work);

Include Files

• mkl_scalapack.h

Description

The p?lantr function returns the value of the 1-norm, or the Frobenius norm, or the infinity norm, or the element of largest absolute value of a trapezoidal or triangular distributed matrix sub(A) = A(ia:ia+m-1, ja:ja+n-1).

Input Parameters

norm  
(global) Specifies what value is returned by the function:

= 'M' or 'm': val = max(abs(Aij)), largest absolute value of the matrix A, it is not a matrix norm.
= '1' or 'O' or 'o': val = norm1(A), 1-norm of the matrix A (maximum column sum),
= 'I' or 'i': val = normI(A), infinity norm of the matrix A (maximum row sum),
= 'F', 'f', 'E' or 'e': val = normF(A), Frobenius norm of the matrix A (square root of sum of squares).

uplo  
(global)
Specifies whether the upper or lower triangular part of the symmetric matrix sub(A) is to be referenced.

= 'U': Upper trapezoidal,
= 'L': Lower trapezoidal.
Note that sub(A) is triangular instead of trapezoidal if \( m = n \).

\[ \text{diag} \]

(global)

Specifies whether the distributed matrix sub(A) has unit diagonal.

- \( \text{='N'} \): Non-unit diagonal.
- \( \text{='U'} \): Unit diagonal.

\[ m \]

(global)

The number of rows in the distributed matrix sub(A). When \( m = 0 \), `p?lantr` is set to zero. \( m \geq 0 \).

\[ n \]

(global)

The number of columns in the distributed matrix sub(A). When \( n = 0 \), `p?lantr` is set to zero. \( n \geq 0 \).

\[ a \]

(local)

Pointer into the local memory to an array of size \( lld_a * \text{LOCc}(ja+n-1) \) containing the local pieces of the distributed matrix sub(A).

\[ ia, ja \]

(global)

The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

\[ \text{desca} \]

(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix A.

\[ \text{work} \]

(local)

Array size \( lwork \).

- \( lwork \geq 0 \) if \( \text{norm} = \text{'M'} \) or \( \text{'m'} \) (not referenced),
- \( nq0 \) if \( \text{norm} = \text{'1'} \), \( \text{'O'} \) or \( \text{'o'} \),
- \( mp0 \) if \( \text{norm} = \text{'I'} \) or \( \text{'i'} \),
- \( 0 \) if \( \text{norm} = \text{'F'} \), \( \text{'F'} \), \( \text{'E'} \) or \( \text{'e'} \) (not referenced),

\( iroffa = \text{mod}(ia-1, mb_a) \), \( icoffa = \text{mod}(ja-1, nb_a) \),

\( \text{iarow} = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow) \),

\( \text{iacol} = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol) \),

\( mp0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow) \),

\( nq0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol) \).

\( \text{indxg2p} \) and \( \text{numroc} \) are ScaLAPACK tool functions; \( \text{myrow} \), \( \text{mycol} \), \( \text{nprow} \), and \( \text{npcol} \) can be determined by calling the function \( \text{blacs_gridinfo} \).

**Output Parameters**

\[ \text{val} \]

The value returned by the function.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.
Applies a permutation matrix to a general distributed matrix, resulting in row or column pivoting.

Syntax

void pslapiv (char *direc, char *rowcol, char *pivroc, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *ip, MKL_INT *jp, MKL_INT *descip, MKL_INT *iwork);

void pdlapiv (char *direc, char *rowcol, char *pivroc, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *ip, MKL_INT *jp, MKL_INT *descip, MKL_INT *iwork);

void pclapiv (char *direc, char *rowcol, char *pivroc, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *ip, MKL_INT *jp, MKL_INT *descip, MKL_INT *iwork);

void pzlapiv (char *direc, char *rowcol, char *pivroc, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *ipiv, MKL_INT *ip, MKL_INT *jp, MKL_INT *descip, MKL_INT *iwork);

Include Files

- mkl_scalapack.h

Description

The p?lapiv function applies either $P$ (permutation matrix indicated by $ipiv$) or $\text{inv}(P)$ to a general $m$-by-$n$ distributed matrix $\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)$, resulting in row or column pivoting. The pivot vector may be distributed across a process row or a column. The pivot vector should be aligned with the distributed matrix $A$. This function will transpose the pivot vector, if necessary.

For example, if the row pivots should be applied to the columns of $\text{sub}(A)$, pass $\text{rowcol} = 'C'$ and $\text{pivroc} = 'C'$.

Input Parameters

direc  (global)
Specifies in which order the permutation is applied:

= 'F' (Forward): Applies pivots forward from top of matrix. Computes $P \cdot \text{sub}(A)$.
= 'B' (Backward): Applies pivots backward from bottom of matrix. Computes $\text{inv}(P) \cdot \text{sub}(A)$.

rowcol  (global)
Specifies if the rows or columns are to be permuted:

= 'R': Rows will be permuted,
= 'C': Columns will be permuted.

pivroc  (global)
Specifies whether $ipiv$ is distributed over a process row or column:

= 'R': $ipiv$ is distributed over a process row,
The number of rows in the distributed matrix sub(A). When \( m = 0 \), `p?lapiv` is set to zero. \( m \geq 0 \).

\( n \) (global)

The number of columns in the distributed matrix sub(A). When \( n = 0 \), `p?lapiv` is set to zero. \( n \geq 0 \).

\( a \) (local)

Pointer into the local memory to an array of size \( lld_a \times \text{LOCc}(ja+n-1) \) containing the local pieces of the distributed matrix sub(A).

\( ia, ja \) (global)

The row and column indices in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix A.

\( \text{ipiv} \) (local)

Array of size \( \text{lipiv} \);

when \( \text{rowcol} = 'R' \) or 'r':

\[ \text{lipiv} \geq \text{LOCr}(ia+m-1) + mb_a \text{ if } \text{pivroc} = 'C' \text{ or } 'c', \]
\[ \text{lipiv} \geq \text{LOCc}(m + \text{mod}(jp-1, nb_p)) \text{ if } \text{pivroc} = 'R' \text{ or } 'r', \text{ and,} \]

when \( \text{rowcol} = 'C' \) or 'c':

\[ \text{lipiv} \geq \text{LOCr}(n + \text{mod}(ip-1, mb_p)) \text{ if } \text{pivroc} = 'C' \text{ or } 'c', \]
\[ \text{lipiv} \geq \text{LOCc}(ja+n-1) + nb_a \text{ if } \text{pivroc} = 'R' \text{ or } 'r'. \]

This array contains the pivoting information. \( \text{ipiv}(i) \) is the global row (column), local row (column) \( i \) was swapped with. When \( \text{rowcol} = 'R' \) or 'r' and \( \text{pivroc} = 'C' \text{ or } 'c' \), or \( \text{rowcol} = 'C' \text{ or } 'c' \text{ and } \text{pivroc} = 'R' \text{ or } 'r' \), the last piece of this array of size \( mb_a \) (resp. \( nb_a \)) is used as workspace. In those cases, this array is tied to the distributed matrix A.

\( \text{ip}, \text{jp} \) (global) The row and column indices in the global matrix \( P \) indicating the first row and the first column of the matrix sub(\( P \)), respectively.

\( \text{descip} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed vector \( \text{ipiv} \).

\( \text{iwork} \) (local).

Array of size \( ldw \), where \( ldw \) is equal to the workspace necessary for transposition, and the storage of the transposed \( \text{ipiv} \): Let \( lcm \) be the least common multiple of \( nprow \) and \( npcol \).

```c
if( *rowcol == 'r' && *pivroc == 'r') {
    if( nprow == npcol) {
        ldw = LOCr( n_p + (*jp-1)%nb_p ) + nb_p;
    } else {
        ...
    }
}
```
Output Parameters

\( a \)

(local).

On exit, the local pieces of the permuted distributed submatrix.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lapv2

Applies a permutation to an \( m \)-by-\( n \) distributed matrix.

Syntax

```
void pslapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT* n, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);
void pdlapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT* n, double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);
void pclapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT* n, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);
void pzlapv2 (const char* direc, const char* rowcol, const MKL_INT* m, const MKL_INT* n, MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca, const MKL_INT* ipiv, const MKL_INT* ip, const MKL_INT* jp, const MKL_INT* descip);
```

Include Files

• mkl_scalapack.h

Description

p?lapv2 applies either \( P \) (permutation matrix indicated by \( ipiv \)) or \( \text{inv}(P) \) to an \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \) denoting \( A(ia:ia+m-1,ja:ja+n-1) \), resulting in row or column pivoting. The pivot vector should be aligned with the distributed matrix \( A \). For pivoting the rows of \( \text{sub}(A) \), \( ipiv \) should be distributed along a process column and replicated over all process rows. Similarly, \( ipiv \) should be distributed along a process row and replicated over all process columns for column pivoting.
Input Parameters

direc (global)
Specifies in which order the permutation is applied:
= 'F' (Forward) Applies pivots Forward from top of matrix. Computes \( P \) sub(\( A \));
= 'B' (Backward) Applies pivots Backward from bottom of matrix. Computes \( \text{inv}( P ) \) * sub(\( A \)).

direct (global)

rowcol (global)
Specifies if the rows or columns are to be permuted:
= 'R' Rows will be permuted,
= 'C' Columns will be permuted.

rowcol (global)

m (global)
The number of rows to be operated on, i.e. the number of rows of the distributed submatrix sub(\( A \)). \( m \geq 0 \).

n (global)
The number of columns to be operated on, i.e. the number of columns of the distributed submatrix sub(\( A \)). \( n \geq 0 \).

a Pointer into local memory to an array of size \( lld_a*LOCc(ja+n-1) \).
On entry, this local array contains the local pieces of the distributed matrix sub(\( A \)) to which the row or columns interchanges will be applied.

ia (global)
The row index in the global array \( a \) indicating the first row of sub(\( A \)).

ja (global)
The column index in the global array \( a \) indicating the first column of sub(\( A \)).

desca (global and local)
Array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

ipiv Array, size \( \geq \text{LOCr}(m_a)+mb_a \) if \( \text{rowcol} = 'R' \), \( \text{LOCc}(n_a)+nb_a \) otherwise.
It contains the pivoting information. \( \text{ipiv}[i-1] \) is the global row (column), local row (column) \( i \) was swapped with. The last piece of the array of size \( mb_a \) or \( nb_a \) is used as workspace. \( \text{ipiv} \) is tied to the distributed matrix \( A \).

ip (global)
The global row index of \( \text{ipiv} \), which points to the beginning of the submatrix on which to operate.

jp (global)
The global column index of \textit{ipiv}, which points to the beginning of the submatrix on which to operate.

\texttt{descip} \\
\text{(global and local)}

Array of size 8.

The array descriptor for the distributed matrix \textit{ipiv}.

\textbf{Output Parameters}

\texttt{a} \\
On exit, this array contains the local pieces of the permuted distributed matrix.

\textbf{p?laqge}

\textit{Scales a general rectangular matrix, using row and column scaling factors computed by \texttt{p?geequ}.}

\textbf{Syntax}

\begin{verbatim}
void pslaqge (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *r, float *c, float *rowcnd, float *colcnd, float *amax, char *equed);
void pdlaqge (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *r, double *c, double *rowcnd, double *colcnd, double *amax, char *equed);
void pclaqge (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *r, float *c, float *rowcnd, float *colcnd, float *amax, char *equed);
void pzlaqge (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *r, double *c, double *rowcnd, double *colcnd, double *amax, char *equed);
\end{verbatim}

\textbf{Include Files}

- \texttt{mkl_scalapack.h}

\textbf{Description}

The \texttt{p?laqge} function equilibrates a general \textit{m}-by-\textit{n} distributed matrix \texttt{sub(A)} = \texttt{A(ia:ia+m-1, ja:ja+n-1)} using the row and scaling factors in the vectors \texttt{r} and \texttt{c} computed by \texttt{p?geequ}.

\textbf{Input Parameters}

\texttt{m} \\
\text{(global)}

The number of rows in the distributed matrix \texttt{sub(A)}. (\textit{m} \geq 0).

\texttt{n} \\
\text{(global)}

The number of columns in the distributed matrix \texttt{sub(A)}. (\textit{n} \geq 0).

\texttt{a} \\
\text{(local)}

Pointer into the local memory to an array of size \texttt{lld_a * LOCc(ja+n-1)}.

On entry, this array contains the distributed matrix \texttt{sub(A)}.
\textit{ia, ja} \hspace{1cm} (global) The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\textit{desca} \hspace{1cm} (global and local) array of size \textit{dlen\_}. The array descriptor for the distributed matrix \( A \).

\textit{r} \hspace{1cm} (local).
Array of size \( \text{LOCr}(m_a) \). The row scale factors for \( \text{sub}(A) \). \( r \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( r \) is tied to the distributed matrix \( A \).

\textit{c} \hspace{1cm} (local).
Array of size \( \text{LOCc}(n_a) \). The row scale factors for \( \text{sub}(A) \). \( c \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( c \) is tied to the distributed matrix \( A \).

\textit{rowcnd} \hspace{1cm} (local).
The global ratio of the smallest \( r[i] \) to the largest \( r[i] \), \( ia-1 \leq i \leq ia+m-2 \).

\textit{colcnd} \hspace{1cm} (local).
The global ratio of the smallest \( c[i] \) to the largest \( c[i] \), \( ia-1 \leq i \leq ia+n-2 \).

\textit{amax} \hspace{1cm} (global).
Absolute value of largest distributed submatrix entry.

**Output Parameters**

\textit{a} \hspace{1cm} (local).
On exit, the equilibrated distributed matrix. See \textit{equed} for the form of the equilibrated distributed submatrix.

\textit{equed} \hspace{1cm} (global)
Specifies the form of equilibration that was done.

- 'N': No equilibration
- 'R': Row equilibration, that is, \( \text{sub}(A) \) has been pre-multiplied by \( \text{diag}(r[ia-1:ia+m-2]) \),
- 'C': column equilibration, that is, \( \text{sub}(A) \) has been post-multiplied by \( \text{diag}(c[ja-1:ja+n-2]) \),
- 'B': Both row and column equilibration, that is, \( \text{sub}(A) \) has been replaced by \( \text{diag}(r[ia-1:ia+m-2])* \text{sub}(A) * \text{diag}(c[ja-1:ja+n-2]) \).

**See Also**

Overview for details of ScalAPACK array descriptor structures and related notations.

\textit{p?laqr0}
Computes the eigenvalues of a Hessenberg matrix and optionally returns the matrices from the Schur decomposition.
Syntax
void pslaqr0(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi, float* h, MKL_INT* desch, float* wr, float* wi, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* descz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info, MKL_INT* reclevel);
void pdlaqr0(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi, double* h, MKL_INT* desch, double* wr, double* wi, MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* descz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info, MKL_INT* reclevel);

Include Files
• mkl_scalapack.h

Description
p?laqr0 computes the eigenvalues of a Hessenberg matrix \( H \) and, optionally, the matrices \( T \) and \( Z \) from the Schur decomposition \( H = Z*T*Z^T \), where \( T \) is an upper quasi-triangular matrix (the Schur form), and \( Z \) is the orthogonal matrix of Schur vectors.

Optionally \( Z \) may be postmultiplied into an input orthogonal matrix \( Q \) so that this function can give the Schur factorization of a matrix \( A \) which has been reduced to the Hessenberg form \( H \) by the orthogonal matrix \( Q \): \( A = Q*H*Q^T = (QZ)*T*(QZ)^T \).

Input Parameters
wantt (global )
Non-zero : the full Schur form \( T \) is required;
Zero : only eigenvalues are required.

wantz (global )
Non-zero : the matrix of Schur vectors \( Z \) is required;
Zero : Schur vectors are not required.

n (global )
The order of the Hessenberg matrix \( H \) (and \( Z \) if wantz is non-zero). \( n \geq 0 \).

ilo, ihi (global )
It is assumed that the matrix \( H \) is already upper triangular in rows and columns \( 1:ilo-1 \) and \( ihi+1:n \). ilo and ihi are normally set by a previous call to p?gebal, and then passed to p?gehrd when the matrix output by ihi is reduced to Hessenberg form. Otherwise ilo and ihi should be set to 1 and \( n \), respectively. If \( n > 0 \), then \( 1 \leq ilo \leq ihi \leq n \).
If \( n = 0 \), then \( ilo = 1 \) and \( ihi = 0 \).

h (global ) array of size \( ld_h * LOC_c(n) \)
The upper Hessenberg matrix \( H \).

desch (global and local )
Array of size \( dlen_\) .
The array descriptor for the distributed matrix \( H \).
**Specify the rows of the matrix** $Z$ **to which transformations must be applied if** $\text{wantz}$ **is non-zero,** $1 \leq \text{ilo} \leq \text{iloz} \leq \text{ihi} \leq \text{ihiz} \leq n.$

**Array of size** $\text{lda}_z \times \text{LOC}(n)$.

If $\text{wantz}$ **is non-zero,** contains the matrix $Z$.

If $\text{wantz}$ **equals zero,** $z$ is not referenced.

**(global and local)** array of size $\text{dlen}_z$.

The array descriptor for the distributed matrix $Z$.

**(local workspace)** array of size $\text{lwork}$

The length of the workspace array $\text{work}$.

**Output Parameters**

On exit, if $\text{wantt}$ **is non-zero,** the matrix $H$ is upper quasi-triangular in rows and columns $\text{ilo}$: $\text{ihi}$, with 1-by-1 and 2-by-2 blocks on the main diagonal. The 2-by-2 diagonal blocks (corresponding to complex conjugate pairs of eigenvalues) are returned in standard form, with $H(i,i) = H(i+1,i+1)$ and $H(i+1,i)^*H(i,i+1) < 0$. If $\text{info} = 0$ and $\text{wantt}$ **equals zero,** the contents of $h$ are unspecified on exit.

The real and imaginary parts, respectively, of the computed eigenvalues $\text{ilo}$ to $\text{ihi}$ are stored in the corresponding elements of $\text{wr}$ and $\text{wi}$. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of $\text{wr}$ and $\text{wi}$, say the $i$-th and $(i+1)$-th, with $\text{wi}[i-1] > 0$ and $\text{wi}[i] < 0$. If $\text{wantt}$ **is non-zero,** the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in $h$.

Updated matrix with transformations applied only to the submatrix $Z(\text{ilo}:\text{ihi},\text{ilo}:\text{ihi})$.

If $\text{COMPZ} = 'I'$, on exit, if $\text{info} = 0$, $z$ contains the orthogonal matrix $Z$ of the Schur vectors of $H$.

If $\text{wantz}$ **is non-zero,** then $Z(\text{ilo}:\text{ihi},\text{iloz}:\text{ihiz})$ is replaced by $Z(\text{ilo}:\text{ihi},\text{iloz}:\text{ihiz})^*U$, when $U$ is the orthogonal/unitary Schur factor of $H(\text{ilo}:\text{ihi},\text{ilo}:\text{ihi})$.

If $\text{wantz}$ **equals zero,** then $z$ is not defined.

On exit, if $\text{info} = 0$, $\text{work}[0]$ **returns the optimal** $\text{lwork}$.

On exit, if $\text{info} = 0$, $\text{iwork}[0]$ **returns the optimal** $\text{liwork}$. 
info

> 0: if info = i, then the function failed to compute all the eigenvalues.
Elements 0:ilo-2 and i:n-1 of wr and wi contain those eigenvalues which have been successfully computed.

> 0: if wantt equals zero, then the remaining unconverged eigenvalues are the eigenvalues of the upper Hessenberg matrix rows and columns ilo through ihi of the final output value of H.

> 0: if wantt is non-zero, then (initial value of H)*U = U*(final value of H), where U is an orthogonal/unitary matrix. The final value of H is upper Hessenberg and quasi-triangular/triangular in rows and columns info+1 through ihi.

> 0: if wantz is non-zero, then (final value of Z(ilo:ihi,iloz:ihi))=U*(initial value of Z(ilo:ihi,iloz:ihi)), where U is the orthogonal/unitary matrix in the previous expression (regardless of the value of wantt).

> 0: if wantz equals zero, then z is not accessed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr1

Sets a scalar multiple of the first column of the product of a 2-by-2 or 3-by-3 matrix and specified shifts.

Syntax

void pslaqr1(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi, float* a, MKL_INT* desca, float* wr, float* wi, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* descz, float* work, MKL_INT* lwork, MKL_INT* ilwork, MKL_INT* info);
void pdlaqr1(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ilo, MKL_INT* ihi, double* a, MKL_INT* desca, double* wr, double* wi, MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* descz, double* work, MKL_INT* lwork, MKL_INT* ilwork, MKL_INT* ilwork, MKL_INT* info);

Include Files

• mkl_scalapack.h

Description

p?laqr1 is an auxiliary function used to find the Schur decomposition and/or eigenvalues of a matrix already in Hessenberg form from columns ilo to ihi.

This is a modified version of p?lahqr from ScaLAPACK version 1.7.3. The following modifications were made:

• Workspace query functionality was added.
• Aggressive early deflation is implemented.
• Aggressive deflation (looking for two consecutive small subdiagonal elements by PSLACONSB) is abandoned.
• The returned Schur form is now in canonical form, i.e., the returned 2-by-2 blocks really correspond to complex conjugate pairs of eigenvalues.
• For some reason, the original version of p?lahqr sometimes did not read out the converged eigenvalues correctly. This is now fixed.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wantt</td>
<td>(global) Non-zero: the full Schur form ( T ) is required; Zero: only eigenvalues are required.</td>
</tr>
<tr>
<td>wantz</td>
<td>(global) Non-zero: the matrix of Schur vectors ( Z ) is required; Zero: Schur vectors are not required.</td>
</tr>
<tr>
<td>n</td>
<td>(global) The order of the Hessenberg matrix ( A ) (and ( Z ) if wantz is non-zero). ( n \geq 0 ).</td>
</tr>
<tr>
<td>ilo, ihi</td>
<td>(global) It is assumed that the matrix ( A ) is already upper quasi-triangular in rows and columns ( ihi+1:n ), and that ( A(ilo,ilo-1) = 0 ) (unless ( ilo = 1 )). p?laqr1 works primarily with the Hessenberg submatrix in rows and columns ( ilo ) to ( ihi ), but applies transformations to all of ( H ) if wantt is non-zero. ( 1 \leq ilo \leq max(1,ihi); ihi \leq n ).</td>
</tr>
<tr>
<td>a</td>
<td>(global) array of size ( lld_a * LOC_c(n) ) On entry, the upper Hessenberg matrix ( A ).</td>
</tr>
<tr>
<td>desca</td>
<td>(global and local) array of size ( dlen_ ). The array descriptor for the distributed matrix ( A ).</td>
</tr>
<tr>
<td>iloz, ihiz</td>
<td>(global) Specify the rows of the matrix ( Z ) to which transformations must be applied if wantz is non-zero. ( 1 \leq iloz \leq ilo; ihi \leq ihiz \leq n ).</td>
</tr>
<tr>
<td>z</td>
<td>(global) array of size ( lld_z * LOC_c(n) ). If wantz is non-zero, on entry ( z ) must contain the current matrix ( Z ) of transformations accumulated by p?hseqr If wantz is zero, ( z ) is not referenced.</td>
</tr>
</tbody>
</table>
descz (global and local) array of size dlen_.
The array descriptor for the distributed matrix Z.

work (local output) array of size lwork

lwork (local)
The size of the work array (lwork≥1).
If lwork=-1, then a workspace query is assumed.

iwork (global and local) array of size ilwork
This holds the some of the IBLK integer arrays.
ilwork (local)
The size of the iwork array (ilwork≥3).

OUTPUT Parameters

a If wantt is non-zero, the matrix A is upper quasi-triangular in rows and columns ilo:ihi, with any 2-by-2 or larger diagonal blocks not yet in standard form. If wantt equals zero, the contents of a are unspecified on exit.

wr, wi (global replicated) array of size n
The real and imaginary parts, respectively, of the computed eigenvalues ilo to ihi are stored in the corresponding elements of wr and wi. If two eigenvalues are computed as a complex conjugate pair, they are stored in consecutive elements of wr and wi, say the i-th and (i+1)th, with wi[i-1] > 0 and wi[i] < 0. If wantt is non-zero, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in a. a may be returned with larger diagonal blocks until the next release.

z On exit z is updated; transformations are applied only to the submatrix Z(ilo:ihiz,ilo:ihi).
If wantz equals zero, z is not referenced.

work[0] On exit, if info = 0, work[0] returns the optimal lwork.

info (global)
< 0: parameter number -info incorrect or inconsistent
= 0: successful exit
> 0: p?laqr1 failed to compute all the eigenvalues ilo to ihi in a total of 30*(ihi-ilo+1) iterations; if info = i, elements i:ihi-1 of wr and wi contain those eigenvalues which have been successfully computed.

Application Notes
This algorithm is very similar to p?ahqr. Unlike p?lahqr, instead of sending one double shift through the largest unreduced submatrix, this algorithm sends multiple double shifts and spaces them apart so that there can be parallelism across several processor row/columns. Another critical difference is that this algorithm aggregates multiple transforms together in order to apply them in a block fashion.

Current Notes and/or Restrictions:
• This code requires the distributed block size to be square and at least six (6); unlike simpler codes like LU, this algorithm is extremely sensitive to block size. Unwise choices of too small a block size can lead to bad performance.
• This code requires \(a\) and \(z\) to be distributed identically and have identical contexts.
• This release currently does not have a function for resolving the Schur blocks into regular 2x2 form after this code is completed. Because of this, a significant performance impact is required while the deflation is done by sometimes a single column of processors.
• This code does not currently block the initial transforms so that none of the rows or columns for any bulge are completed until all are started. To offset pipeline start-up it is recommended that at least \(2 \times \text{LCM}(\text{NPROW}, \text{NPCOL})\) bulges are used (if possible)
• The maximum number of bulges currently supported is fixed at 32. In future versions this will be limited only by the incoming work array.
• The matrix \(A\) must be in upper Hessenberg form. If elements below the subdiagonal are nonzero, the resulting transforms may be nonsimilar. This is also true with the LAPACK function.
• For this release, it is assumed \(rsrc_ = csrc_ = 0\)
• Currently, all the eigenvalues are distributed to all the nodes. Future releases will probably distribute the eigenvalues by the column partitioning.
• The internals of this function are subject to change.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{p?laqr2}

\textit{Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).}

Syntax

\begin{verbatim}
void pslaqr2(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, float* a, MKL_INT* desca, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* descz,
MKL_INT* ns, MKL_INT* nd, float* sr, float* si, float* t, MKL_INT* ldt, float* v,
MKL_INT* ldv, float* wr, float* wi, float* work, MKL_INT* lwork);
void pdlaqr2(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot,
MKL_INT* nw, double* a, MKL_INT* desca, MKL_INT* iloz, MKL_INT* ihiz, double* z,
MKL_INT* descz, MKL_INT* ns, MKL_INT* nd, double* sr, double* si, double* t, MKL_INT* ldt,
double* v, MKL_INT* ldv, double* wr, double* wi, double* work, MKL_INT* lwork);
\end{verbatim}

Include Files

• mkl_scalapack.h

Description

\texttt{p?laqr2} accepts as input an upper Hessenberg matrix \(A\) and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output \(A\) is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of \(A\). It is to be hoped that the final version of \(A\) has many zero subdiagonal entries.

This function handles small deflation windows which is affordable by one processor. Normally, it is called by \texttt{p?laqr1}. All the inputs are assumed to be valid without checking.
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for
optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and
SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or
effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-
dependent optimizations in this product are intended for use with Intel microprocessors. Certain
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to
the applicable product User and Reference Guides for more information regarding the specific instruction
sets covered by this notice.

Notice revision #20110804

Input Parameters

wantt  (global )
If wantt is non-zero, then the Hessenberg matrix \( A \) is fully updated so that
the quasi-triangular Schur factor may be computed (in cooperation with the
calling function).
If wantt equals zero, then only enough of \( A \) is updated to preserve the
eigenvalues.

wantz  (global )
If wantz is non-zero, then the orthogonal matrix \( Z \) is updated so that the
orthogonal Schur factor may be computed (in cooperation with the calling
function).
If wantz equals zero, then \( z \) is not referenced.

n  (global )
The order of the matrix \( A \) and (if wantz is non-zero) the order of the
orthogonal matrix \( Z \).

ktop, kbot  (global )
It is assumed without a check that either \( kbot = n \) or \( A(kbot+1,kbot)=0 \).
kbot and ktop together determine an isolated block along the diagonal of
the Hessenberg matrix. However, \( A(ktop,ktop-1)=0 \) is not essentially
necessary if wantt is non-zero.

nw  (global )
Deflation window size. \( 1 \leq nw \leq (kbot-ktop+1) \). Normally \( nw \geq 3 \) if p?laqr2 is
called by p?laqr1.

a  (local ) array of size \( ld_a \times LOC(c(n)) \)
The initial \( n \)-by-\( n \) section of \( a \) stores the Hessenberg matrix undergoing
aggressive early deflation.

desca  (global and local) array of size \( dlen_\).
The array descriptor for the distributed matrix \( A \).

iloz, ihiz  (global )
Specify the rows of the matrix \( Z \) to which transformations must be applied if \( \text{wantz} \) is non-zero. \( 1 \leq \text{iloz} \leq \text{ihiz} \leq n \).

\[ z \]

Array of size \( \text{ldd}_z \times \text{LOC}(n) \)

If \( \text{wantz} \) is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix \( Z(\text{iloz}:\text{ihiz}, \ kbot: \text{ktop}) \), stored in \( z \), from the right.

If \( \text{wantz} \) is zero, then \( z \) is unreferenced.

\[ \text{descz} \]

(global and local) array of size \( \text{dlen}_z \).

The array descriptor for the distributed matrix \( Z \).

\[ t \]

(local workspace) array of size \( \text{ldt} \times \text{nw} \).

\[ \text{ldt} \]

(local)

The leading dimension of the array \( t \). \( \text{ldt} \geq \text{nw} \).

\[ v \]

(local workspace) array of size \( \text{ldv} \times \text{nw} \).

\[ \text{ldv} \]

(local)

The leading dimension of the array \( v \). \( \text{ldv} \geq \text{nw} \).

\[ \text{wr, wi} \]

(local workspace) array of size \( \text{kbot} \).

\[ \text{work} \]

(local workspace) array of size \( \text{lwork} \).

\[ \text{lwork} \]

(local)

\( \text{work(} \text{lwork} \text{)} \) is a local array and \( \text{lwork} \) is assumed big enough so that \( \text{lwork} \geq \text{nw} \times \text{nw} \).

**OUTPUT Parameters**

\[ a \]

On output \( a \) has been transformed by an orthogonal similarity transformation, perturbed, and returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.

\[ z \]

(global)

The number of unconverged (that is, approximate) eigenvalues returned in \( \text{sr} \) and \( \text{si} \) that may be used as shifts by the calling function.

\[ \text{nd} \]

(global)

The number of converged eigenvalues uncovered by this function.

\[ \text{sr, si} \]

(global) array of size \( \text{kbot} \)

On output, the real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in \( \text{sr}[\text{kbot}-\text{nd}-\text{ns}] \) through \( \text{sr}[\text{kbot}-\text{nd}-1] \) and \( \text{si}[\text{kbot}-\text{nd}-\text{ns}] \) through \( \text{si}[\text{kbot}-\text{nd}-1] \), respectively.
On processor #0, the real and imaginary parts of converged eigenvalues are stored in $sr[kbot-nd]$ through $sr[kbot-1]$ and $si[kbot-nd]$ through $si[kbot-1]$, respectively. On other processors, these entries are set to zero.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laqr3
Performs the orthogonal/unitary similarity transformation of a Hessenberg matrix to detect and deflate fully converged eigenvalues from a trailing principal submatrix (aggressive early deflation).

Syntax
void pslaqr3(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nw, float* h, MKL_INT* desch, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* descz, MKL_INT* ns, MKL_INT* nd, float* sr, float* si, float* v, MKL_INT* descv, MKL_INT* nh, float* t, MKL_INT* desct, MKL_INT* nv, float* wv, MKL_INT* descw, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* reclevel);

void pdlaqr3(MKL_INT* wantt, MKL_INT* wantz, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nw, double* h, MKL_INT* desch, MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* descz, MKL_INT* ns, MKL_INT* nd, double* sr, double* si, double* v, MKL_INT* descv, MKL_INT* nh, double* t, MKL_INT* desct, MKL_INT* nv, double* wv, MKL_INT* descw, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* reclevel);

Include Files
• mkl_scalapack.h

Description
This function accepts as input an upper Hessenberg matrix $H$ and performs an orthogonal similarity transformation designed to detect and deflate fully converged eigenvalues from a trailing principal submatrix. On output $H$ is overwritten by a new Hessenberg matrix that is a perturbation of an orthogonal similarity transformation of $H$. It is to be hoped that the final version of $H$ has many zero subdiagonal entries.

Input Parameters
wantt

(global )

If wantt is non-zero, then the Hessenberg matrix $H$ is fully updated so that the quasi-triangular Schur factor may be computed (in cooperation with the calling function).

If wantt equals zero, then only enough of $H$ is updated to preserve the eigenvalues.

wantz

(global )

If wantz is non-zero, then the orthogonal matrix $Z$ is updated so that the orthogonal Schur factor may be computed (in cooperation with the calling function).

If wantz equals zero, then $Z$ is not referenced.
The order of the matrix $H$ and (if $\text{wantz}$ is non-zero), the order of the orthogonal matrix $Z$.

It is assumed that either $k\text{top} = 1$ or $H(k\text{top},k\text{top}-1)=0$. $k\text{bot}$ and $k\text{top}$ together determine an isolated block along the diagonal of the Hessenberg matrix.

It is assumed without a check that either $k\text{bot} = n$ or $H(k\text{bot}+1,k\text{bot})=0$. $k\text{bot}$ and $k\text{top}$ together determine an isolated block along the diagonal of the Hessenberg matrix.

Deflation window size. $1 \leq nw \leq (k\text{bot}-k\text{top}+1)$.

The initial $n$-by-$n$ section of $H$ stores the Hessenberg matrix undergoing aggressive early deflation.

The array descriptor for the distributed matrix $H$.

Specify the rows of the matrix $Z$ to which transformations must be applied if $\text{wantz}$ is non-zero. $1 \leq \text{iloz}\leq \text{ihiz}\leq n$.

Array of size $lld_z \ast LOC_c(n)$

If $\text{wantz}$ is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix $Z(\text{iloz}:\text{ihiz},k\text{bot}:k\text{top})$ from the right.

If $\text{wantz}$ is zero, then $z$ is unreferenced.

The array descriptor for the distributed matrix $Z$.

An $nw$-by-$nw$ distributed work array.

The array descriptor for the distributed matrix $V$.

The number of columns of $t$. $nh \geq nw$.

The array descriptor for the distributed matrix $T$. 

The array descriptor for the distributed matrix $V$. 

The array descriptor for the distributed matrix $T$. 


The number of rows of work array `wv` available for workspace. `nv\geq nw`.

`wv`  
(global workspace) array of size `lld_w*LOC_\_`(`nw`)  

`descw`  
(global and local) array of size `dlen_`_.  
The array descriptor for the distributed matrix `wv`.  

`work`  
(local workspace) array of size `lwork`.  

`lwork`  
(local)  
The size of the work array `work (lwork\geq 1)`. `lwork = 2* nw` suffices, but greater efficiency may result from larger values of `lwork`. If `lwork = -1`, then a workspace query is assumed; `p?laqr3` only estimates the optimal workspace size for the given values of `n`, `nw`, `ktop` and `kbot`. The estimate is returned in `work[0]`. No error message related to `lwork` is issued by `xerbla`. Neither `h` nor `z` are accessed.  

`iwork`  
(local workspace) array of size `liwork` 

`liwork`  
(local)  
The length of the workspace array `iwork (liwork\geq 1)`. If `liwork=-1`, then a workspace query is assumed. 

**OUTPUT Parameters**

`h`  
On output `h` has been transformed by an orthogonal similarity transformation, perturbed, and the returned to Hessenberg form that (it is to be hoped) has some zero subdiagonal entries.  

`z`  
IF `wantz` is non-zero, then on output, the orthogonal similarity transformation mentioned above has been accumulated into the matrix `Z (iloz:ihiz, kbot:ktop)` from the right. If `wantz` is zero, then `z` is unreferenced.  

`ns`  
(global)  
The number of unconverged (that is, approximate) eigenvalues returned in `sr` and `si` that may be used as shifts by the calling function.  

`nd`  
(global)  
The number of converged eigenvalues uncovered by this function.  

`sr`, `si`  
(global) array of size `kbot`. The real and imaginary parts of approximate eigenvalues that may be used for shifts are stored in `sr[kbot-nd-ns]` through `sr[kbot-nd-1]` and `si[kbot-nd-ns]` through `si[kbot-nd-1]`, respectively. The real and imaginary parts of converged eigenvalues are stored in `sr[kbot-nd]` through `sr[kbot-1]` and `si[kbot-nd]` through `si[kbot-1]`, respectively.  

`work[0]`  
On exit, if `info = 0`, `work[0]` returns the optimal `lwork`  

`iwork[0]`  
On exit, if `info = 0`, `iwork[0]` returns the optimal `liwork`
See Also for details of ScaLAPACK array descriptor structures and related notations.

p?laqr5
Performs a single small-bulge multi-shift QR sweep.

Syntax

void pslaqr5(mkl_int* wantt, mkl_int* wantz, mkl_int* kacc22, mkl_int* n, mkl_int* ktop, mkl_int* kbot, mkl_int* nshfts, float* sr, float* si, float* h, mkl_int* desch, mkl_int* iloz, mkl_int* ihiz, float* z, mkl_int* descz, float* work, mkl_int* lwork, mkl_int* iwork, mkl_int* liwork);

void pdlaqr5(mkl_int* wantt, mkl_int* wantz, mkl_int* kacc22, mkl_int* n, mkl_int* ktop, mkl_int* kbot, mkl_int* nshfts, double* sr, double* si, double* h, mkl_int* desch, mkl_int* iloz, mkl_int* ihiz, double* z, mkl_int* descz, double* work, mkl_int* lwork, mkl_int* iwork, mkl_int* liwork);

Include Files

• mkl_scalapack.h

Description

This auxiliary function called by p?laqr0 performs a single small-bulge multi-shift QR sweep by chasing separated groups of bulges along the main block diagonal of a Hessenberg matrix $H$.

Input Parameters

wantt

(global) scalar

wantt is non-zero if the quasi-triangular Schur factor is being computed. wantt is set to zero otherwise.

wantz

(global) scalar

wantz is non-zero if the orthogonal Schur factor is being computed. wantz is set to zero otherwise.

kacc22

(global)

Value 0, 1, or 2. Specifies the computation mode of far-from-diagonal orthogonal updates.

= 0: p?laqr5 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.

= 1: p?laqr5 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

= 2: p?laqr5 accumulates reflections, uses matrix-matrix multiply to update the far-from-diagonal matrix entries, and takes advantage of 2-by-2 block structure during matrix multiplies.

n

(global) scalar

The order of the Hessenberg matrix $H$ and, if wantz is non-zero, the order of the orthogonal matrix $Z$.

ktop, kbot

(global) scalar
These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $ktop = 1$ or $H(ktop,ktop-1) = 0$ and either $kbot = n$ or $H(kbot+1,kbot) = 0$.

$nshfts$ (global) scalar

$nshfts$ gives the number of simultaneous shifts. $nshfts$ must be positive and even.

$sr, si$ (global) Array of size $nshfts$

$sr$ contains the real parts and $si$ contains the imaginary parts of the $nshfts$ shifts of origin that define the multi-shift QR sweep.

$h$ (local) Array of size $lld_h * LOC_c(n)$

On input $h$ contains a Hessenberg matrix $H$.

$desc$ (global and local)

array of size $dlen_$. The array descriptor for the distributed matrix $H$.

$iloz, ihiz$ (global)

Specify the rows of the matrix $Z$ to which transformations must be applied if $wantz$ is non-zero. $1 \leq iloz \leq ihiz \leq n$

$z$ (local) array of size $lld_z * LOC_c(n)$

If $wantz$ is non-zero, then the QR Sweep orthogonal similarity transformation is accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$ from the right. If $wantz$ equals zero, then $z$ is unreferenced.

$descz$ (global and local) array of size $dlen_$. The array descriptor for the distributed matrix $Z$.

$work$ (local workspace) array of size $lwork$

$lwork$ (local)

The size of the $work$ array ($lwork \geq 1$). If $lwork=-1$, then a workspace query is assumed.

$liwork$ (local)

The size of the $iwork$ array ($liwork \geq 1$). If $liwork=-1$, then a workspace query is assumed.

Output Parameters

$h$ A multi-shift QR sweep with shifts $sr(j)+i*si(j)$ is applied to the isolated diagonal block in rows and columns $ktop$ through $kbot$ of the matrix $H$. 

ScaLAPACK Routines
If `wantz` is non-zero, `z` is updated with transformations applied only to the submatrix $Z(ilo:ihiz,kbot:ktop)$.

On exit, if $\text{info} = 0$, `work[0]` returns the optimal $lwork$.

On exit, if $\text{info} = 0$, `iwork[0]` returns the optimal $liwork$.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?laqsy**

Scales a symmetric/Hermitian matrix, using scaling factors computed by `p?poequ`.

**Syntax**

```c
void pslaqsy(char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *sr, float *sc, float *scond, float *amax, char *equed);
void pdlaqsy(char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *sr, double *sc, double *scond, double *amax, char *equed);
void pclaqsy(char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *sr, float *sc, float *scond, float *amax, char *equed);
void pzlaqsy(char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *sr, double *sc, double *scond, double *amax, char *equed);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?laqsy` function equilibrates a symmetric distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$ using the scaling factors in the vectors $sr$ and $sc$. The scaling factors are computed by `p?poequ`.

**Input Parameters**

- `uplo`
  (global) Specifies the upper or lower triangular part of the symmetric distributed matrix $\text{sub}(A)$ to be referenced:
  - 'U': Upper triangular part;
  - 'L': Lower triangular part.

- `n`
  (global)
  The order of the distributed matrix $\text{sub}(A)$. $n \geq 0$.

- `a`
  (local).
  Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$.
  On entry, this array contains the local pieces of the distributed matrix $\text{sub}(A)$. On entry, the local pieces of the distributed symmetric matrix $\text{sub}(A)$.
If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular part of the matrix, and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced.

If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( \text{sub}(A) \) contains the lower triangular part of the matrix, and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

\( \text{ia}, \text{ja} \)

(globally)

The row and column indices in the global matrix \( A \) indicating the first row and the first column of the matrix \( \text{sub}(A) \), respectively.

\( \text{desca} \)

(globally and locally) array of size \( \text{dlen}_\cdot \). The array descriptor for the distributed matrix \( A \).

\( \text{sr} \)

(local)

Array of size \( \text{LOCr}(m_a) \). The scale factors for the matrix \( A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1) \). \( \text{sr} \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( \text{sr} \) is tied to the distributed matrix \( A \).

\( \text{sc} \)

(local)

Array of size \( \text{LOCc}(m_a) \). The scale factors for the matrix \( A(\text{ia}:\text{ia}+m-1, \text{ja}:\text{ja}+n-1) \). \( \text{sc} \) is aligned with the distributed matrix \( A \), and replicated across every process column. \( \text{sc} \) is tied to the distributed matrix \( A \).

\( \text{scond} \)

(global)

Ratio of the smallest \( \text{sr}[i] \) (respectively \( \text{sc}[j] \)) to the largest \( \text{sr}[i] \) (respectively \( \text{sc}[j] \)), with \( \text{ia} - 1 \leq i < \text{ia} + \text{m} - 1 \) and \( \text{ja} - 1 \leq j < \text{ja} + \text{n} - 1 \).

\( \text{amax} \)

(global)

Absolute value of largest distributed submatrix entry.

**Output Parameters**

\( \text{a} \)

On exit,

if \( \text{equed} = 'Y' \), the equilibrated matrix:

\[
\text{diag}\left(\text{sr}_{\text{ia}}, \ldots, \text{sr}_{\text{ia}+\text{n}-1}\right) \times \text{sub}(A) \times \text{diag}\left(\text{sc}_{\text{ja}}, \ldots, \text{sc}_{\text{ja}+\text{n}-1}\right).
\]

\( \text{equed} \)

(global)

Specifies whether or not equilibration was done.

\( = 'N' \): No equilibration.

\( = 'Y' \): Equilibration was done, that is, \( \text{sub}(A) \) has been replaced by:

\[
\text{diag}\left(\text{sr}_{\text{ia}}, \ldots, \text{sr}_{\text{ia}+\text{n}-1}\right) \times \text{sub}(A) \times \text{diag}\left(\text{sc}_{\text{ja}}, \ldots, \text{sc}_{\text{ja}+\text{n}-1}\right).
\]

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
p?lared1d
Redistributes an array assuming that the input array, bycol, is distributed across rows and that all process columns contain the same copy of bycol.

Syntax

void pslared1d (MKL_INT *n, MKL_INT *ia, MKL_INT *ja, MKL_INT *desc, float *bycol, float *byall, float *work, MKL_INT *lwork);
void pdlared1d (MKL_INT *n, MKL_INT *ia, MKL_INT *ja, MKL_INT *desc, double *bycol, double *byall, double *work, MKL_INT *lwork);

Include Files

- mkl_scalapack.h

Description

The p?lared1d function redistributes a 1D array. It assumes that the input array bycol is distributed across rows and that all process column contain the same copy of bycol. The output array byall is identical on all processes and contains the entire array.

Input Parameters

np = Number of local rows in bycol()

\( n \) (global)
The number of elements to be redistributed. \( n \geq 0 \).

\( ia, ja \) (global) \( ia, ja \) must be equal to 1.

desc (local) array of size 9. A 2D array descriptor, which describes bycol.

bycol (local).
Distributed block cyclic array of global size \( n \) and of local size \( np \). bycol is distributed across the process rows. All process columns are assumed to contain the same value.

work (local).
size \( lwork \). Used to hold the buffers sent from one process to another.

lwork (local)
The size of the work array. \( lwork \geq \text{numroc}(n, \text{desc}[nb_], 0, 0, npcol) \).

Output Parameters

byall (global).
Global size \( n \), local size \( n \). byall is exactly duplicated on all processes. It contains the same values as bycol, but it is replicated across all processes rather than being distributed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
p?lared2d
Redistributes an array assuming that the input array
byrow is distributed across columns and that all
process rows contain the same copy of byrow.

Syntax
void pslared2d (MKL_INT *n, MKL_INT *ia, MKL_INT *ja, MKL_INT *desc, float *byrow, 
float *byall, float *work, MKL_INT *lwork);
void pdlared2d (MKL_INT *n, MKL_INT *ia, MKL_INT *ja, MKL_INT *desc, double *byrow, double *byall, double *work, MKL_INT *lwork);

Include Files
• mkl_scalapack.h

Description
The p?lared2d function redistributes a 1D array. It assumes that the input array byrow is distributed across
columns and that all process rows contain the same copy of byrow. The output array byall will be identical
on all processes and will contain the entire array.

Input Parameters
np = Number of local rows in byrow()

  n (global)
  The number of elements to be redistributed. \( n \geq 0 \).

  ia, ja (global) ia, ja must be equal to 1.

  desc (local) array of size dlen_. A 2D array descriptor, which describes byrow.

  byrow (local).
  Distributed block cyclic array of global size \( n \) and of local size \( np \).
  byrow is distributed across the process columns. All process rows are assumed to contain the same value.

  work (local).
  size lwork. Used to hold the buffers sent from one process to another.

  lwork (local) The size of the work array. \( lwork \geq \text{numroc}(n, \text{desc}[\text{nb}], 0, 0, \text{npcol}) \).

Output Parameters
byall (global).
Global size \( n \), local size \( n \). byall is exactly duplicated on all processes. It contains the same values as byrow, but it is replicated across all processes rather than being distributed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?larf**

Applies an elementary reflector to a general rectangular matrix.

**Syntax**

```c
void pslarf (char *side, MKL_INT *m, MKL_INT *n, float *v, MKL_INT *iv, MKL_INT *jc, MKL_INT *descv, float *incv, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work);

void pdlarf (char *side, MKL_INT *m, MKL_INT *n, double *v, MKL_INT *iv, MKL_INT *jc, MKL_INT *descv, MKL_INT *incv, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work);

void pclarf (char *side, MKL_INT *m, MKL_INT *n, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *incv, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work);

void pzlarf (char *side, MKL_INT *m, MKL_INT *n, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jc, MKL_INT *descv, MKL_INT *incv, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?larf function applies a real/complex elementary reflector $Q$ (or $Q^T$) to a real/complex $m$-by-$n$ distributed matrix $\text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1)$, from either the left or the right. $Q$ is represented in the form

$$Q = I - \tau * v * v',$$

where $\tau$ is a real/complex scalar and $v$ is a real/complex vector.

If $\tau = 0$, then $Q$ is taken to be the unit matrix.

**Input Parameters**

- **side**
  - (global).
  - = 'L': form $Q^{*}\text{sub}(C)$,
  - = 'R': form $\text{sub}(C)^{*}Q$, $Q=Q^{T}$.

- **m**
  - (global)
  - The number of rows in the distributed submatrix $\text{sub}(A)$. ($m \geq 0$).

- **n**
  - (global)
  - The number of columns in the distributed submatrix $\text{sub}(A)$. ($n \geq 0$).

- **v**
  - (local).
  - Pointer into the local memory to an array of size $\text{ld}_v * \text{LOC}(n, v)$, containing the local pieces of the global distributed matrix $V$ representing the Householder transformation $Q$,
  - $V(iv:iv+m-1, jv)$ if $\text{side} = 'L$ and $\text{incv} = 1$,
The array \( V \) is the representation of \( Q \). \( V \) is not used if \( \tau = 0 \).

**iv, jv**  
(global) The row and column indices in the global matrix \( V \) indicating the first row and the first column of the matrix sub(\( V \)), respectively.

**descv**  
(global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( V \).

**incv**  
(global)  
The global increment for the elements of \( V \). Only two values of \( incv \) are supported in this version, namely 1 and \( m_v \).

\( incv \) must not be zero.

**tau**  
(local).  
Array of size \( LOCc(jv) \) if \( incv = 1 \), and \( LOCr(iv) \) otherwise. This array contains the Householder scalars related to the Householder vectors.  
\( tau \) is tied to the distributed matrix \( V \).

**c**  
(local).  
Pointer into the local memory to an array of size \( lld_c * LOCc(jc+n-1) \), containing the local pieces of sub(\( C \)).

**ic, jc**  
(global)  
The row and column indices in the global matrix \( C \) indicating the first row and the first column of the matrix sub(\( C \)), respectively.

**descc**  
(global and local) array of size \( dlen_ \)_. The array descriptor for the distributed matrix \( C \).

**work**  
(local).  
Array of size \( lwork \).

If \( incv = 1 \),  
if \( side = 'L' \),  
if \( ivcol = iccol \),  
\( lwork \geq nqc0 \) else  
\( lwork \geq mpc0 + \max(1, nqc0) \) end if  
else if \( side = 'R' \),  
\( lwork \geq nqc0 + \max(\max(1, mpc0), numroc(numroc(n+icoffc,nb_v,0,0,npc0),nb_v,0,0,lcmq)) \) end if  
else if \( incv = m_v \),
if \( \text{side} = 'L', \)

\[
\text{lwork} \geq \text{mpc} 0 + \max\left( \max(1, nqc 0 ), \text{numroc}\left( \text{numroc}(m + \text{iroffc}, \text{mb}_v, 0, 0, \text{nprow}), \text{mb}_v, 0, 0, \text{lcmp}) \right) \right)
\]

else if \( \text{side} = 'R', \)

if \( \text{ivrow} = \text{icrow}, \)

\[
\text{lwork} \geq \text{mpc} 0
\]

else

\[
\text{lwork} \geq nqc 0 + \max(1, \text{mpc} 0 )
\]

end if

end if

end if,

where \( \text{lcm} \) is the least common multiple of \( \text{nprow} \) and \( \text{npcol} \) and \( \text{lcm} = \text{ilcm}(\text{nprow}, \text{npcol}), \text{lcmp} = \text{lcm}/\text{nprow}, \text{lcmq} = \text{lcm}/\text{npcol}, \text{iroffc} = \text{mod}(\text{ic}-1, \text{mb}_c), \text{icoffc} = \text{mod}(\text{jc}-1, \text{nb}_c), \text{icrow} = \text{indxg2p}(\text{ic}, \text{mb}_c, \text{myrow}, \text{rsrC}_C, \text{nprow}), \text{iccol} = \text{indxg2p}(\text{jc}, \text{nb}_c, \text{mycol}, \text{csrC}_C, \text{npcol}), \text{mpc} 0 = \text{numroc}(m + \text{iroffc}, \text{mb}_c, \text{myrow}, \text{icrow}, \text{nprow}), \text{nqc} 0 = \text{numroc}(n + \text{icoffc}, \text{nb}_c, \text{mycol}, \text{iccol}, \text{npcol}), \text{ilcm}, \text{indxg2p}, \text{and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo.}\)

Output Parameters

c

(local).

On exit, \( \text{sub}(C) \) is overwritten by the \( Q^{*}\text{sub}(C) \) if \( \text{side} = 'L', \) or \( \text{sub}(C) * Q \) if \( \text{side} = 'R'. \)

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\text{p?larfb}

Applies a block reflector or its transpose/conjugate-transpose to a general rectangular matrix.

Syntax

void pslarfb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, float *t, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work);

do pack (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, double *t, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work);
void pclarfb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex8 *t, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *desc, MKL_Complex8 *work );
void pzlarfb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex16 *t, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *desc, MKL_Complex16 *work );

Include Files
- mkl_scalapack.h

Description
The p?larfb function applies a real/complex block reflector $Q$ or its transpose $Q^T$/conjugate transpose $Q^H$ to a real/complex distributed $m$-by-$n$ matrix sub($C$) = $C(ic:ic+m-1, jc:jc+n-1)$ from the left or the right.

Input Parameters

side (global)
- If $side$ = 'L': apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the Left;
- If $side$ = 'R': apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the Right.

trans (global)
- If $trans$ = 'N': no transpose, apply $Q$;
  - For real flavors, if $trans$='T': transpose, apply $Q^T$;
  - For complex flavors, if $trans$ = 'C': conjugate transpose, apply $Q^H$;

direct (global) Indicates how $Q$ is formed from a product of elementary reflectors.
- If $direct$ = 'F': $Q = H(1)*H(2)*...*H(k)$ (Forward)
- If $direct$ = 'B': $Q = H(k)*...*H(2)*H(1)$ (Backward)

storev (global)
- Indicates how the vectors that define the elementary reflectors are stored:
  - If $storev$ = 'C': Columnwise
  - If $storev$ = 'R': Rowwise.

m (global)
The number of rows in the distributed matrix sub($C$). ($m \geq 0$).

n (global)
The number of columns in the distributed matrix sub($C$). ($n \geq 0$).

k (global)
The order of the matrix $T$.

v (local).
Pointer into the local memory to an array of size
It contains the local pieces of the distributed vectors $V$ representing the Householder transformation.

If $storev = 'C'$ and $\text{side} = 'L'$,
$$\text{lld}_v \geq \max(1, \text{LOCr}(iv+m-1))$$
if $storev = 'C'$ and $\text{side} = 'R'$,
$$\text{lld}_v \geq \max(1, \text{LOCr}(iv+n-1))$$
if $storev = 'R'$, $\text{lld}_v \geq \text{LOCr}(jv+k-1)$.

($iv, jv$)
The row and column indices in the global matrix $V$ indicating the first row and the first column of the matrix sub($V$), respectively.

($\text{descv}$)
(global and local) array of size $dlen_\cdot$. The array descriptor for the distributed matrix $V$.

($c$)
(local).

Pointer into the local memory to an array of size $\text{lld}_c \ast \text{LOCc}(jc+n-1)$, containing the local pieces of sub($C$).

($ic, jc$)
(global) The row and column indices in the global matrix $C$ indicating the first row and the first column of the matrix sub($C$), respectively.

($\text{descc}$)
(global and local) array of size $dlen_\cdot$. The array descriptor for the distributed matrix $C$.

($\text{work}$)
(local).
Workspace array of size $lwork$.
If $storev = 'C'$,
if $\text{side} = 'L'$,
$$lwork \geq ( nqc0 + mpc0 ) \ast k$$
else if $\text{side} = 'R'$,
$$lwork \geq ( nqc0 + \max( npv0 + \text{numroc}( numroc( n + icoffc, nb_v, 0, 0, npcol ), nb_v, 0, 0, lcmq ), mpc0 ) ) \ast k$$
end if
else if $\text{storev} = 'R'$,
if $\text{side} = 'L'$,
$$lwork \geq ( mpc0 + \max( mqv0 + \text{numroc}( numroc( m + iroffc, mb_v, 0, 0, nprow ), mb_v, 0, 0, lcm ) ), nqc0 ) ) \ast k$$
else if $\text{side} = 'R'$,
$$lwork \geq ( mpc0 + nqc0 ) \ast k$$
end if
end if,

where

\[ \text{lcmq} = \frac{lcm}{ncol} \] with \[ lcm = \text{iclm}(nprow, ncol) \],

\[ \text{iroffv} = \text{mod}(iv-1, mb_v), \text{icoffv} = \text{mod}(jv-1, nb_v), \]

\[ \text{ivrow} = \text{indxg2p}(iv, mb_v, myrow, rsrcc_v, nprow), \]

\[ \text{ivcol} = \text{indxg2p}(jv, nb_v, mycol, csrcc_v, ncol), \]

\[ \text{MqV0} = \text{numroc}(m+icoffv, nb_v, mycol, ivcol, ncol), \]

\[ \text{NpV0} = \text{numroc}(n+iroffv, mb_v, myrow, ivrow, nrow), \]

\[ \text{iroffc} = \text{mod}(ic-1, mb_c), \text{icoffc} = \text{mod}(jc-1, nb_c), \]

\[ \text{icrow} = \text{indxg2p}(ic, mb_c, myrow, rsrcc_c, nprow), \]

\[ \text{iccol} = \text{indxg2p}(jc, nb_c, mycol, csrcc_c, ncol), \]

\[ \text{MpC0} = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nrow), \]

\[ \text{NpC0} = \text{numroc}(n+icoffc, mb_c, myrow, icrow, nrow), \]

\[ \text{NqC0} = \text{numroc}(n+icoffc, nb_c, mycol, iccol, ncol), \]

\[ \text{l lcm}, \text{ indxg2p}, \text{ and } \text{numroc} \text{ are ScaLAPACK tool functions; } \text{myrow}, \text{ mycol}, \text{ npro w}, \text{ and } \text{ncol} \text{ can be determined by calling the function } \text{blacs_gridinfo}. \]

**Output Parameters**

\( t \) (local).

Array of size \( mb_v * mb_vif \) if \( \text{storev} = \text{'R'} \), and \( nb_v * nb_vif \) if \( \text{storev} = \text{'C'} \). The triangular matrix \( t \) is the representation of the block reflector.

\( c \) (local).

On exit, sub(C) is overwritten by the \( Q^* \text{sub}(C) \), or \( Q'^* \text{sub}(C) \), or \( \text{sub}(C) * Q \), or \( \text{sub}(C) * Q' \). \( Q' \) is transpose (conjugate transpose) of \( Q \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?larfc**

Applies the conjugate transpose of an elementary reflector to a general matrix.

**Syntax**

```c
void pclarfc (char *side, MKL_INT *m, MKL_INT *n, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_INT *incv, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jic, MKL_INT *descic, MKL_Complex8 *work);
void pzlarfc (char *side, MKL_INT *m, MKL_INT *n, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_INT *incv, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jic, MKL_INT *descic, MKL_Complex16 *work);
```
Include Files
• mkl_scalapack.h

Description
The `p?larfc` function applies a complex elementary reflector $QH$ to a complex $m$-by-$n$ distributed matrix $\text{sub}(C) = C(\text{ic:ic+m-1, jc:jc+n-1})$, from either the left or the right. $Q$ is represented in the form

$$Q = i - \tau v^*v',$$

where $\tau$ is a complex scalar and $v$ is a complex vector.

If $\tau = 0$, then $Q$ is taken to be the unit matrix.

Input Parameters

- **side** (global)
  - if `side = 'L'`: form $QH*\text{sub}(C)$;
  - if `side = 'R'`: form $\text{sub}(C)*QH$.

- **m** (global)
  The number of rows in the distributed matrix $\text{sub}(C)$. ($m \geq 0$).

- **n** (global)
  The number of columns in the distributed matrix $\text{sub}(C)$. ($n \geq 0$).

- **v** (local)
  Pointer into the local memory to an array of size $\text{lld}_v * \text{LOCc}(n_v)$, containing the local pieces of the global distributed matrix $V$ representing the Householder transformation $Q$,

  - $V(\text{iv:iv+m-1, jv})$ if `side = 'L' and incv = 1`,
  - $V(\text{iv, jv:iv+m-1})$ if `side = 'L' and incv = $m_v$,
  - $V(\text{iv:iv+n-1, jv})$ if `side = 'R' and incv = 1`,
  - $V(\text{iv, jv:jv+n-1})$ if `side = 'R' and incv = $m_v$.

  The array $v$ is the representation of $Q$. $v$ is not used if $\tau = 0$.

- **iv, jv** (global)
  The row and column indices in the global matrix $V$ indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.

- **descv** (global and local) array of size $\text{dlen}_v$. The array descriptor for the distributed matrix $V$.

- **incv** (global)
  The global increment for the elements of $v$. Only two values of $\text{incv}$ are supported in this version, namely 1 and $m_v$.

  $\text{incv}$ must not be zero.

- **tau** (local)
  Array of size $\text{LOCc}(jv)$ if $\text{incv} = 1$, and $\text{LOCr}(iv)$ otherwise. This array contains the Householder scalars related to the Householder vectors.
\( \tau \) is tied to the distributed matrix \( V \).

\( \mathbf{c} \) (local).

Pointer into the local memory to an array of size \( \text{lld}_\mathbf{c} * \text{LOC}c(j+c+n-1) \), containing the local pieces of \( \text{sub}(C) \).

\( \mathbf{ic}, \mathbf{jc} \) (global)

The row and column indices in the global matrix \( C \) indicating the first row and the first column of the matrix \( \text{sub}(C) \), respectively.

\( \text{descc} \) (global and local) array of size \( \text{dlen}_\mathbf{c} \). The array descriptor for the distributed matrix \( C \).

\( \text{work} \) (local).

Workspace array of size \( \text{lwork} \).

If \( \text{incv} = 1 \),

if \( \text{side} = \text{`L'} \),

if \( \text{ivcol} = \text{icc} \),

\( \text{lwork} \geq nqc0 \)

else

\( \text{lwork} \geq \text{mpc}0 + \max( 1, nqc0 ) \)

end if

else if \( \text{side} = \text{`R'} \),

\( \text{lwork} \geq nqc0 + \max( \max( 1, \text{mpc}0 ), \text{numroc}( \text{numroc}( \text{n}+\text{icoffc},\text{nb}_v,0,0,\text{npcol} ), \text{nb}_v,0,0,\text{lcmq} ) ) \)

end if

else if \( \text{incv} = \text{m}_\mathbf{v} \),

if \( \text{side} = \text{`L'} \),

\( \text{lwork} \geq \text{mpc}0 + \max( \max( 1, nqc0 ), \text{numroc}( \text{numroc}( \text{m}+\text{iroffc},\text{mb}_v,0,0,\text{nprow} ),\text{mb}_v,0,0,\text{lcmp} ) ) \)

else if \( \text{side} = \text{`R'} \),

if \( \text{ivrow} = \text{icrow} \),

\( \text{lwork} \geq \text{mpc}0 \)

else

\( \text{lwork} \geq nqc0 + \max( 1, \text{mpc}0 ) \)

end if

end if

end if,

where \( \text{lcm} \) is the least common multiple of \( \text{nprow} \) and \( \text{npcol} \) and \( \text{lcm} = \text{ilcm}(\text{nprow}, \text{npcol}) \),

\( \text{lcmp} = \text{lcm}/\text{nprow}, \text{lcmq} = \text{lcm}/\text{npcol}, \)

\( \text{iroffc} = \text{mod}(\text{i}-1, \text{mb}_c), \text{icoffc} = \text{mod}(\text{j}+\text{c}+\text{n}-1, \text{nb}_c), \)

\( \text{lwork} \geq \text{nqc}0 + \max( 1, \text{mpc}0 ) \)
icrow = indxg2p(ic, mb_c, myrow, rsr, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrc_c, npcol),
mr0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),
nr0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo.

Output Parameters

c (local).
On exit, sub(C) is overwritten by the $Q^H \cdot sub(C)$ if side = 'L', or sub(C) * $Q^H$ if side = 'R'.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larfg
Generates an elementary reflector (Householder matrix).

Syntax
void pslarfg (MKL_INT *n, float *alpha, MKL_INT *iax, MKL_INT *jax, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, float *tau);
void pdlarfg (MKL_INT *n, double *alpha, MKL_INT *iax, MKL_INT *jax, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, double *tau);
void pclarfg (MKL_INT *n, MKL_Complex8 *alpha, MKL_INT *iax, MKL_INT *jax, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, MKL_Complex8 *tau);
void pzlarfg (MKL_INT *n, MKL_Complex16 *alpha, MKL_INT *iax, MKL_INT *jax, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, MKL_Complex16 *tau);

Include Files
• mkl_scalapack.h

Description
The p?larfg function generates a real/complex elementary reflector $H$ of order $n$, such that

$$H \cdot sub(X) = H \cdot \begin{pmatrix} x(ix, jx) \\ \alpha \end{pmatrix} = \begin{pmatrix} \alpha \\ 0 \end{pmatrix}, \quad H^* \cdot H = I,$$

where $\alpha$ is a scalar (a real scalar - for complex flavors), and sub(X) is an $(n-1)$-element real/complex distributed vector $X(ix:ix+n-2, jx)$ if incx = 1 and $X(ix, jx:jx+n-2)$ if incx = m_x. $H$ is represented in the form
where \( \tau \) is a real/complex scalar and \( v \) is a real/complex \((n-1)\)-element vector. Note that \( H \) is not Hermitian.

If the elements of \( \text{sub}(X) \) are all zero (and \( X(iax, jax) \) is real for complex flavors), then \( \tau = 0 \) and \( H \) is taken to be the unit matrix.

Otherwise \( 1 \leq \text{real}(\tau) \leq 2 \) and \( \text{abs}(\tau-1) \leq 1 \).

**Input Parameters**

\( n \)  
(global)  
The global order of the elementary reflector. \( n \geq 0 \).

\( iax, jax \)  
(global)  
The global row and column indices of \( X(iax, jax) \) in the global matrix \( X \).

\( x \)  
(local).  
Pointer into the local memory to an array of size \( lld_x * \text{LOCc}(n_x) \). This array contains the local pieces of the distributed vector \( \text{sub}(X) \). Before entry, the incremented array \( \text{sub}(X) \) must contain vector \( x \).

\( ix, jx \)  
(global)  
The row and column indices in the global matrix \( X \) indicating the first row and the first column of \( \text{sub}(X) \), respectively.

\( \text{descx} \)  
(global and local)  
Array of size \( dlen_\_ \). The array descriptor for the distributed matrix \( X \).

\( incx \)  
(global)  
The global increment for the elements of \( x \). Only two values of \( incx \) are supported in this version, namely 1 and \( m_\_x \). \( incx \) must not be zero.

**Output Parameters**

\( \alpha \)  
(local)  
On exit, \( \alpha \) is computed in the process scope having the vector \( \text{sub}(X) \).

\( x \)  
(local).  
On exit, it is overwritten with the vector \( v \).

\( \tau \)  
(local).  
Array of size \( \text{LOCc}(jx) \) if \( incx = 1 \), and \( \text{LOCr}(ix) \) otherwise. This array contains the Householder scalars related to the Householder vectors. \( \tau \) is tied to the distributed matrix \( X \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?larft**

Forms the triangular vector $T$ of a block reflector $H=I-V^*TV$. 

**Syntax**

```c
void pslarft (char *direct, char *storev, MKL_INT *n, MKL_INT *k, float *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, float *tau, float *t, float *work);
void pdlarft (char *direct, char *storev, MKL_INT *n, MKL_INT *k, double *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, double *tau, double *t, double *work);
void pclarft (char *direct, char *storev, MKL_INT *n, MKL_INT *k, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex8 *tau, MKL_Complex8 *t, MKL_Complex8 *work);
void pzlarft (char *direct, char *storev, MKL_INT *n, MKL_INT *k, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex16 *tau, MKL_Complex16 *t, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?larft function forms the triangular factor $T$ of a real/complex block reflector $H$ of order $n$, which is defined as a product of $k$ elementary reflectors.

If `direct = 'F'`, $H = H(1)H(2)\ldots H(k)$, and $T$ is upper triangular;

If `direct = 'B'`, $H = H(k)H(2)H(1)$, and $T$ is lower triangular.

If `storev = 'C'`, the vector which defines the elementary reflector $H(i)$ is stored in the $i$-th column of the distributed matrix $V$, and $H = I-V^*TV$.

If `storev = 'R'`, the vector which defines the elementary reflector $H(i)$ is stored in the $i$-th row of the distributed matrix $V$, and $H = I-V^*TV$.

**Input Parameters**

- **direct**
  
  Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
  
  - if `direct = 'F'`: $H = H(1)H(2)\ldots H(k)$ (forward)
  - if `direct = 'B'`: $H = H(k)\ldots H(2)H(1)$ (backward).

- **storev**
  
  Specifies how the vectors that define the elementary reflectors are stored (See Application Notes below):
  
  - if `storev = 'C'`: columnwise;
  - if `storev = 'R'`: rowwise.
The order of the block reflector $H$. $n \geq 0$.

The order of the triangular factor $T_i$ is equal to the number of elementary reflectors.

$1 \leq k \leq mb_v (= nb_v)$.

Pointer into the local memory to an array of local size

$\text{LOCr}(iv+n-1) \times \text{LOCc}(jv+k-1)$ if $\text{storev} = 'C'$, and

$\text{LOCr}(iv+k-1) \times \text{LOCc}(jv+n-1)$ if $\text{storev} = 'R'$.

The distributed matrix $V$ contains the Householder vectors. (See Application Notes below).

The row and column indices in the global matrix $V$ indicating the first row and the first column of the matrix $\text{sub}(V)$, respectively.

Array of size $\text{LOCr}(iv+k-1)$ if $\text{incv} = m_v$, and $\text{LOCc}(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors.

$\text{tau}$ is tied to the distributed matrix $V$.

Workspace array of size $k^*(k -1)/2$.

Array of size $nb_v \times nb_v$ if $\text{storev} = 'C'$, and $mb_v \times mb_v$ otherwise. It contains the $k$-by-$k$ triangular factor of the block reflector associated with $v$.

If $\text{direct} = 'F'$, $\text{t}$ is upper triangular;

if $\text{direct} = 'B'$, $\text{t}$ is lower triangular.

The shape of the matrix $V$ and the storage of the vectors that define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?larz**

 Applies an elementary reflector as returned by **p?tzrzf** to a general matrix.

**Syntax**

```c
void pslarz (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l,
            float *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv,
            MKL_INT *incv, float *tau, float *c, MKL_INT *ic,
            MKL_INT *jc, MKL_INT *descc, float *work);

void pdlarz (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l,
            double *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv,
            MKL_INT *incv, double *tau, double *c, MKL_INT *ic,
            MKL_INT *jc, MKL_INT *descc, double *work);

void pclarz (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l,
            MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv,
            MKL_INT *descv, MKL_INT *incv, MKL_Complex8 *tau,
            MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc,
            MKL_INT *descc, MKL_Complex8 *work);

void pzlarz (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l,
            MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv,
            MKL_INT *descv, MKL_INT *incv, MKL_Complex16 *tau,
            MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc,
            MKL_INT *descc, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The **p?larz** function applies a real/complex elementary reflector $Q$ (or $Q^T$) to a real/complex $m$-by-$n$ distributed matrix $\text{sub}(C) = C(\text{ic:ic+m-1}, \text{jc:jc+n-1})$, from either the left or the right. $Q$ is represented in the form

$$Q = I - \tau v v^\top,$$

where $\tau$ is a real/complex scalar and $v$ is a real/complex vector.

If $\tau = 0$, then $Q$ is taken to be the unit matrix.

$Q$ is a product of $k$ elementary reflectors as returned by **p?tzrzf**.
Input Parameters

\( \text{side} \) (global)

if \( \text{side} = 'L' \): form \( Q^* \text{sub}(C) \),
if \( \text{side} = 'R' \): form \( \text{sub}(C)^*Q \), \( Q = Q^* \) (for real flavors).

\( m \) (global)

The number of rows in the distributed matrix \( \text{sub}(C) \). \( m \geq 0 \).

\( n \) (global)

The number of columns in the distributed matrix \( \text{sub}(C) \). \( n \geq 0 \).

\( l \) (global)

The columns of the distributed matrix \( \text{sub}(A) \) containing the meaningful part of the Householder reflectors. If \( \text{side} = 'L' \), \( m \geq l \geq 0 \), if \( \text{side} = 'R' \), \( n \geq l \geq 0 \).

\( v \) (local).

Pointer into the local memory to an array of size \( \text{ld}_d \times \text{LOCc}(n_v) \) containing the local pieces of the global distributed matrix \( V \) representing the Householder transformation \( Q \),
\( V(iv:iv+l-1, jv) \) if \( \text{side} = 'L' \) and \( \text{incv} = 1 \),
\( V(iv, jv:jv+l-1) \) if \( \text{side} = 'L' \) and \( \text{incv} = m_v \),
\( V(iv:iv+l-1, jv) \) if \( \text{side} = 'R' \) and \( \text{incv} = 1 \),
\( V(iv, jv:jv+l-1) \) if \( \text{side} = 'R' \) and \( \text{incv} = m_v \).

The vector \( v \) in the representation of \( Q \). \( v \) is not used if \( \tau = 0 \).

\( iv, jv \) (global) The row and column indices in the global distributed matrix \( V \) indicating the first row and the first column of the matrix \( \text{sub}(V) \), respectively.

descv (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( V \).

\( \text{incv} \) (global)

The global increment for the elements of \( V \). Only two values of \( \text{incv} \) are supported in this version, namely 1 and \( m_v \).

\( \text{incv} \) must not be zero.

\( \tau \) (local)

Array of size \( \text{LOCc}(jv) \) if \( \text{incv} = 1 \), and \( \text{LOCr}(iv) \) otherwise. This array contains the Householder scalars related to the Householder vectors. \( \tau \) is tied to the distributed matrix \( V \).

\( c \) (local).

Pointer into the local memory to an array of size \( \text{ld}_c \times \text{LOCc}(j_c+n-1) \), containing the local pieces of \( \text{sub}(C) \).

\( ic, j_c \) (global)
The row and column indices in the global matrix $C$ indicating the first row and the first column of the matrix sub($C$), respectively.

$\text{descC}$

global and local) array of size $\text{dlen}_\epsilon$. The array descriptor for the distributed matrix $C$.

$\text{work}$

(local).

Array of size $\text{lw}_{\text{ork}}$

If $\text{inc}_{\text{v}} = 1$,

if $\text{side} = 'L$',

if $\text{iv}_{\text{col}} = \text{ic}_{\text{col}}$,

$\text{lw}_{\text{ork}} \geq Nq_{C0}$

else

$\text{lw}_{\text{ork}} \geq Mp_{C0} + \text{max}(1, Nq_{C0})$

end if

else if $\text{side} = 'R$',

$\text{lw}_{\text{ork}} \geq Nq_{C0} + \text{max}(\text{max}(1, Mp_{C0}), \text{numroc}(\text{numroc}(n+ico_{ffc}, nb_{v}, 0,0,npc_{ol}), nb_{v}, 0,0,lcmq))$

end if

else if $\text{inc}_{\text{v}} = m_{v}$,

if $\text{side} = 'L$',

$\text{lw}_{\text{ork}} \geq Mp_{C0} + \text{max}(\text{max}(1, Nq_{C0}), \text{numroc}(\text{numroc}(m+iro_{ffc}, mb_{v}, 0,0,npw_{r}, mb_{v}, 0,0,lcmp)))$

else if $\text{side} = 'R$',

if $\text{iv}_{\text{row}} = \text{ic}_{\text{row}}$,

$\text{lw}_{\text{ork}} \geq Mp_{C0}$

else

$\text{lw}_{\text{ork}} \geq Nq_{C0} + \text{max}(1, Mp_{C0})$

end if

end if

end if.

Here $lcm$ is the least common multiple of $\text{npro}_{w}$ and $\text{npc}_{ol}$ and

$lcm = \text{ilcm}(\text{npro}_{w}, \text{npc}_{ol}), lcm_{\text{p}} = lcm / \text{npro}_{w},$

$lcmq = lcm / \text{npc}_{ol},$

$iro_{ffc} = \text{mod}(ic-1, mb_{c}), ico_{ffc} = \text{mod}(jc-1, nb_{c})$, $ic_{\text{row}} = \text{indxg2p}(ic, mb_{c}, \text{my}_{\text{row}}, rsr_{c}_{c}, npw_{r})$, $ic_{\text{col}} = \text{indxg2p}(jc, nb_{c}, \text{my}_{\text{col}}, csr_{c}_{c}, npc_{ol})$, $mpc_{0} = \text{numroc}(m+iro_{ffc}, mb_{c}, \text{my}_{\text{row}}, ic_{\text{row}}, npw_{r})$, $nqc_{0} = \text{numroc}(n+ico_{ffc}, nb_{c}, \text{my}_{\text{col}}, ic_{\text{col}}, npc_{ol})$,
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; *myrow, *mycol, *nprow, and *npcol can be determined by calling the function blacs_gridinfo.

Output Parameters

\( c \)  

(local).

On exit, sub(\( C \)) is overwritten by the \( Q^{*} \) sub(\( C \)) if \( \text{side} = 'L' \), or sub(\( C \)) * \( Q \) if \( \text{side} = 'R' \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?larzb**

Applies a block reflector or its transpose/conjugate-transpose as returned by **p?tzrzf** to a general matrix.

Syntax

```c
void pslarzb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *l, float *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, float *t, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work );
void pdlarzb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *l, double *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, double *t, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work );
void pclarzb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *l, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex8 *t, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work );
void pzlarzb (char *side, char *trans, char *direct, char *storev, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_INT *l, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_Complex16 *t, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description

The **p?larzb** function applies a real/complex block reflector \( Q \) or its transpose \( Q^T \) (conjugate transpose \( Q^H \) for complex flavors) to a real/complex distributed \( m \)-by-\( n \) matrix sub(\( C \)) = \( C(ic:ic+m-1, jc:jc+n-1) \) from the left or the right.

\( Q \) is a product of \( k \) elementary reflectors as returned by **p?tzrzf**.

Currently, only \( \text{storev} = 'R' \) and \( \text{direct} = 'B' \) are supported.
Input Parameters

side  
(if global)
if side = 'L': apply Q or $Q^T$ ($Q^H$ for complex flavors) from the Left;
if side = 'R': apply Q or $Q^T$ ($Q^H$ for complex flavors) from the Right.

trans  
(if global)
if trans = 'N': No transpose, apply Q;
If trans='T': Transpose, apply $Q^T$ (real flavors);
If trans='C': Conjugate transpose, apply $Q^H$ (complex flavors).

direct  
(if global)
Indicates how $H$ is formed from a product of elementary reflectors.
if direct = 'F': $H = H(1)*H(2)*...*H(k) - forward (not supported) ;
if direct = 'B': $H = H(k)*...*H(2)*H(1) - backward.$

storev  
(if global)
Indicates how the vectors that define the elementary reflectors are stored:
if storev = 'C': columnwise (not supported ).
if storev = 'R': rowwise.

m  
(if global)
The number of rows in the distributed submatrix sub(C). ($m \geq 0$).

n  
(if global)
The number of columns in the distributed submatrix sub(C). ($n \geq 0$).

k  
(if global)
The order of the matrix $T$. (= the number of elementary reflectors whose product defines the block reflector).

l  
(if global)
The columns of the distributed submatrix sub(A) containing the meaningful part of the Householder reflectors.
If side = 'L', $m \geq l \geq 0$,
if side = 'R', $n \geq l \geq 0$.

v  
(if local).
Pointer into the local memory to an array of size $lld_v * LOCr(jv+m-1)$ if side = 'L', $lld_v * LOCC(jv+m-1)$ if side = 'R'.
It contains the local pieces of the distributed vectors $V$ representing the Householder transformation as returned by p?tzrzf.

$lld_v \geq LOCr(iv+k-1)$.

iv, jv  
(if global)
The row and column indices in the global matrix $V$ indicating the first row and the first column of the submatrix sub($V$), respectively.

$\text{descv}$ (global and local) array of size $dlen_$. The array descriptor for the distributed matrix $V$.

$t$ (local)

Array of size $mb_v*mb_v$.

The lower triangular matrix $T$ in the representation of the block reflector.

$c$ (local).

Pointer into the local memory to an array of size $lld_c * \text{LOCc}(jc+n-1)$.

On entry, the $m$-by-$n$ distributed matrix sub(C).

$ic, jc$ (global)

The row and column indices in the global matrix $C$ indicating the first row and the first column of the submatrix sub(C), respectively.

$\text{descc}$ (global and local) array of size $dlen_$. The array descriptor for the distributed matrix $C$.

$\text{work}$ (local).

Array of size $lwork$.

If $\text{storev} = 'C'$,

if $\text{side} = 'L'$,

$lwork \geq (nqc0 + mpc0) * k$

else if $\text{side} = 'R'$,

$lwork \geq (nqc0 + \max(npv0 + \text{numroc}(n+icoffc, nb_v, 0, 0, npcol), nb_v, 0, 0, lcmq), mpc0)) * k$

end if

else if $\text{storev} = 'R'$,

if $\text{side} = 'L'$,

$lwork \geq (mpc0 + \max(mqv0 + \text{numroc}(m+iroffc, mb_v, 0, 0, nprow), mb_v, 0, 0, lcmq), nqc0)) * k$

else if $\text{side} = 'R'$,

$lwork \geq (mpc0 + nqc0) * k$

end if

end if.

Here $lcmq = lcm/npcol$ with $lcm = iclm(nprow, npc0)$,

$\text{iroffv} = \text{mod}(iv-1, mb_v), \text{icoffv} = \text{mod}(jv-1, nb_v)$,

$\text{ivrow} = \text{indxg2p}(iv, mb_v, myrow, rsrc_v, nprow),$. 

1605
ivcol = indxg2p(jv, nb_v, mycol, csrcc_v, npcol),
mqv0 = numroc(m+icoffv, nb_v, mycol, ivcol, npcol),
npv0 = numroc(n+iroffv, mb_v, myrow, ivrow, nprow),
iroffc = mod(ic-1, mb_c), icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrcc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csrcc_c, npcol),
mpc0 = numroc(m+iroffc, mb_c, myrow, icrow, nprow),
npc0 = numroc(n+icoffc, mb_c, myrow, icrow, nprow),
nqc0 = numroc(n+icoffc, nb_c, mycol, iccol, npcol),
ilcm, indxg2p, and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo.

Output Parameters

\( c \) (local).
On exit, sub(C) is overwritten by the \( Q^*\text{sub(C)} \), or \( Q'^*\text{sub(C)} \), or \( \text{sub(C)}*Q \), or \( \text{sub(C)}*Q' \), where \( Q' \) is the transpose (conjugate transpose) of \( Q \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{p?larzc}
Applies (multiplies by) the conjugate transpose of an elementary reflector as returned by \texttt{p?tzrzf} to a general matrix.

Syntax

\begin{verbatim}
void pclarzc (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l, MKL_Complex8 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_INT *incv, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work);
\end{verbatim}

\begin{verbatim}
void pzlarzc (char *side, MKL_INT *m, MKL_INT *n, MKL_INT *l, MKL_Complex16 *v, MKL_INT *iv, MKL_INT *jv, MKL_INT *descv, MKL_INT *incv, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work);
\end{verbatim}

Include Files

- mkl_scalapack.h

Description

The \texttt{p?larzc} function applies a complex elementary reflector \( Q^H \) to a complex \( m \)-by-\( n \) distributed matrix \( \text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1) \), from either the left or the right. \( Q \) is represented in the form
\[ Q = i - \tau \cdot v^Ho, \]
where \( \tau \) is a complex scalar and \( v \) is a complex vector.
If \( \tau = 0 \), then \( Q \) is taken to be the unit matrix.
Q is a product of \( k \) elementary reflectors as returned by \( \text{p?tzrf} \).

**Input Parameters**

\( \text{side} \) (global)

- if \( \text{side} = 'L' \): form \( Q^H \text{sub}(C) \);
- if \( \text{side} = 'R' \): form \( \text{sub}(C)Q^H \).

\( m \) (global)

The number of rows in the distributed matrix \( \text{sub}(C) \). \( (m \geq 0) \).

\( n \) (global)

The number of columns in the distributed matrix \( \text{sub}(C) \). \( (n \geq 0) \).

\( l \) (global)

The columns of the distributed matrix \( \text{sub}(A) \) containing the meaningful part of the Householder reflectors.

- If \( \text{side} = 'L' \), \( m \geq l \geq 0 \),
- if \( \text{side} = 'R' \), \( n \geq l \geq 0 \).

\( v \) (local).

Pointer into the local memory to an array of size \( \text{ld}_v \times \text{LOCc}(n_v) \) containing the local pieces of the global distributed matrix \( V \) representing the Householder transformation \( Q \),

- \( V(\text{iv:iv}+l-1, \text{jv}) \) if \( \text{side} = 'L' \) and \( \text{incv} = 1 \),
- \( V(\text{iv}, \text{jv:jv}+l-1) \) if \( \text{side} = 'L' \) and \( \text{incv} = m_v \),
- \( V(\text{iv:iv}+l-1, \text{jv}) \) if \( \text{side} = 'R' \) and \( \text{incv} = 1 \),
- \( V(\text{iv}, \text{jv:jv}+l-1) \) if \( \text{side} = 'R' \) and \( \text{incv} = m_v \).

The vector \( v \) in the representation of \( Q \). \( v \) is not used if \( \tau = 0 \).

\( \text{iv}, \text{jv} \) (global)

The row and column indices in the global matrix \( V \) indicating the first row and the first column of the matrix \( \text{sub}(V) \), respectively.

\( \text{descv} \) (global and local) array of size \( dlen_\)_. The array descriptor for the distributed matrix \( V \).

\( \text{incv} \) (global)

The global increment for the elements of \( V \). Only two values of \( \text{incv} \) are supported in this version, namely 1 and \( m_v \).

\( \text{incv} \) must not be zero.

\( \tau \) (local)

Array of size \( \text{LOCc}(jv) \) if \( \text{incv} = 1 \), and \( \text{LOCr}(iv) \) otherwise. This array contains the Householder scalars related to the Householder vectors.

\( \tau \) is tied to the distributed matrix \( V \).

\( c \) (local).
Pointer into the local memory to an array of size $\text{lld}_c \times \text{LOCc}(\text{jc} + n - 1)$, containing the local pieces of sub($C$).

$ic, jc$

The row and column indices in the global matrix $C$ indicating the first row and the first column of the matrix sub($C$), respectively.

$\text{descc}$ (global and local) array of size $dlen_\text{c}$. The array descriptor for the distributed matrix $C$.

$work$ (local).

If $incv = 1$,
if $side = 'L'$,
if $ivcol = iccol$,
$\text{lwork} \geq nqc0$
else
$\text{lwork} \geq mpc0 + \max(1, nqc0)$
end if
else if $side = 'R'$,
$\text{lwork} \geq nqc0 + \max(1, mpc0), \text{numroc(numroc} (n+icoffc, nb_v, 0, 0, npcol), nb_v, 0, 0, lcmq))$  end if
else if $incv = m_v$,
if $side = 'L'$,
$\text{lwork} \geq mpc0 + \max(1, nqc0), \text{numroc(numroc} (m+iroffc, mb_v, 0, 0, nprow), mb_v, 0, 0, lcm))$
else if $side = 'R'$,
if $ivrow = icrow$,
$\text{lwork} \geq mpc0$
else
$\text{lwork} \geq nqc0 + \max(1, mpc0)$
end if
end if
end if

Here $lcm$ is the least common multiple of $nprow$ and $npcol$;

$lcm = \text{ilcm}(nprow, npcol), lcm = lcm/nprow, lcmq = lcm/npcol,$
$\text{iroffc} = \text{mod}(ic-1, mb_c), \text{icoffc} = \text{mod}(jc-1, nb_c),$ 
$\text{icrow} = \text{indxg2p}(ic, mb_c, myrow, rsrcc_c, nprow),$ 
$\text{iccol} = \text{indxg2p}(jc, nb_c, mycol, csrrc_c, npcol),$ 
$\text{mpc0} = \text{numroc}(m+iroffc, mb_c, myrow, icrow, nprow),$ 
$nqc0 = \text{numroc}(n+icoffc, nb_c, mycol, iccol, npcol),$ 
$\text{lcm}, \text{indxg2p}, \text{and \ numroc are ScaLAPACK tool functions;}$

$myrow, mycol, nprow, \text{ and npcol can be determined by calling the function blacs_gridinfo}.$

Output Parameters

$c$ (local).
On exit, sub(C) is overwritten by the $Q^H \times \text{sub(C)}$ if side = 'L', or $\text{sub(C)} \times Q^H$ if side = 'R'.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?larzt
Forms the triangular factor $T$ of a block reflector $H = I - V^T \times T \times V^H$ as returned by p?tzrzf.

Syntax

```c
void pslarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , float *v , MKL_INT *iv , MKL_INT *descv , float *tau , float *t , float *work );
void pdlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , double *v , MKL_INT *iv , MKL_INT *descv , double *tau , double *t , double *work );
void pclarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex8 *v , MKL_INT *iv , MKL_INT *descv , MKL_Complex8 *tau , MKL_Complex8 *t , MKL_Complex8 *work );
void pzlarzt (char *direct , char *storev , MKL_INT *n , MKL_INT *k , MKL_Complex16 *v , MKL_INT *iv , MKL_INT *descv , MKL_Complex16 *tau , MKL_Complex16 *t , MKL_Complex16 *work );
```

Include Files

- mkl_scalapack.h

Description
The p?larzt function forms the triangular factor $T$ of a real/complex block reflector $H$ of order greater than $n$, which is defined as a product of $k$ elementary reflectors as returned by p?tzrzf.

If direct = 'F', $H = H(1) \times H(2) \times \ldots \times H(k)$, and $T$ is upper triangular;
If direct = 'B', $H = H(k) \times \ldots \times H(2) \times H(1)$, and $T$ is lower triangular.

If storev = 'C', the vector which defines the elementary reflector $H(i)$, is stored in the $i$-th column of the array $v$, and $H = i-v \times t \times v'$.
If storev = 'R', the vector, which defines the elementary reflector $H(i)$, is stored in the $i$-th row of the array $v$, and $H = i-v^T \times t \times v$

Currently, only storev = 'R' and direct = 'B' are supported.

Input Parameters

direct
(global)
Specifies the order in which the elementary reflectors are multiplied to form the block reflector:
if direct = 'F': $H = H(1) \times H(2) \times \ldots \times H(k)$ (Forward, not supported)
if direct = 'B': $H = H(k) \times \ldots \times H(2) \times H(1)$ (Backward).
storev (global)
Specifies how the vectors which defines the elementary reflectors are stored:
if storev = 'C': columnwise (not supported);
if storev = 'R': rowwise.

n (global)
The order of the block reflector $H$. $n \geq 0$.

k (global)
The order of the triangular factor $T$ (= the number of elementary reflectors).
$1 \leq k \leq mb_v (= nb_v)$.

v Pointer into the local memory to an array of local size $\text{LOCr}(iv+k-1) \times \text{LOCc}(jv+n-1)$.
The distributed matrix $V$ contains the Householder vectors. See Application Notes below.

iv, jv (global) The row and column indices in the global matrix $V$ indicating the first row and the first column of the matrix sub($V$), respectively.

descv (local) array of size $dlen_v$. The array descriptor for the distributed matrix $V$.

tau (local)
Array of size $\text{LOCr}(iv+k-1)$ if $\text{incv} = m_v$, and $\text{LOCc}(jv+k-1)$ otherwise. This array contains the Householder scalars related to the Householder vectors.
$\tau$ is tied to the distributed matrix $V$.

work (local).
Workspace array of size $(k*(k-1)/2)$.

Output Parameters

v (local)

Array of size $mb_v*mb_v$. It contains the $k$-by-$k$ triangular factor of the block reflector associated with $v$. $t$ is lower triangular.

Application Notes
The shape of the matrix $V$ and the storage of the vectors which define the $H(i)$ is best illustrated by the following example with $n = 5$ and $k = 3$. The elements equal to 1 are not stored; the corresponding array elements are modified but restored on exit. The rest of the array is not used.
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lascl

Multiplies a general rectangular matrix by a real scalar defined as \( C_{to}/C_{from} \).

Syntax

void pslascl (char *type, float *cfrom, float *cto, MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);

void pdlascl (char *type, double *cfrom, double *cto, MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);

void pclascl (char *type, float *cfrom, float *cto, MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);

void pzlascl (char *type, double *cfrom, double *cto, MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
Include Files

- mkl_scalapack.h

Description

The p?lascl function multiplies the $m$-by-$n$ real/complex distributed matrix sub($A$) denoting $A(ia:ia+m-1, ja:ja+n-1)$ by the real/complex scalar $cto/cfrom$. This is done without over/underflow as long as the final result $cto*A(i,j)/cfrom$ does not over/underflow. $type$ specifies that sub($A$) may be full, upper triangular, lower triangular or upper Hessenberg.

Input Parameters

$\textit{type}$  
*(global)*  
$\textit{type}$ indicates the storage type of the input distributed matrix.  
if $\textit{type} = 'G'$: sub($A$) is a full matrix,  
if $\textit{type} = 'L'$: sub($A$) is a lower triangular matrix,  
if $\textit{type} = 'U'$: sub($A$) is an upper triangular matrix,  
if $\textit{type} = 'H'$: sub($A$) is an upper Hessenberg matrix.

$\textit{cfrom, cto}$  
*(global)*  
The distributed matrix sub($A$) is multiplied by $cto/cfrom$. $A(i,j)$ is computed without over/underflow if the final result $cto*A(i,j)/cfrom$ can be represented without over/underflow. $cfrom$ must be nonzero.

$m$  
*(global)*  
The number of rows in the distributed matrix sub($A$). ($m \geq 0$).

$n$  
*(global)*  
The number of columns in the distributed matrix sub($A$). ($n \geq 0$).

$a$  
*(local input/local output)*  
Pointer into the local memory to an array of size $lld_a * LOCc(ja+n-1)$. This array contains the local pieces of the distributed matrix sub($A$).

$\textit{ia, ja}$  
*(global)*  
The column and row indices in the global matrix $A$ indicating the first row and column of the matrix sub($A$), respectively.

$\textit{desca}$  
*(global and local)*  
Array of size $dlen_$. The array descriptor for the distributed matrix $A$.

Output Parameters

$a$  
*(local)*  
On exit, this array contains the local pieces of the distributed matrix multiplied by $cto/cfrom$.

$\textit{info}$  
*(local)*  
if $\textit{info} = 0$: the execution is successful.
if $info < 0$: If the $i$-th argument is an array and the $j$-th entry, indexed
$j-1$, had an illegal value, then $info = -(i*100+j)$,
if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lase2

Initializes an $m$-by-$n$ distributed matrix.

Syntax
void pslase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const float* alpha,
const float* beta, float* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca);
void pdlase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const double* alpha,
const double* beta, double* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca);
void pclase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const MKL_Complex8* alpha,
const MKL_Complex8* beta, MKL_Complex8* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca);
void pzlase2 (const char* uplo, const MKL_INT* m, const MKL_INT* n, const MKL_Complex16* alpha,
const MKL_Complex16* beta, MKL_Complex16* a, const MKL_INT* ia, const MKL_INT* ja, const MKL_INT* desca);

Include Files
• mkl_scalapack.h

Description
p?lase2 initializes an $m$-by-$n$ distributed matrix sub( $A$ ) denoting $A(ia:ia+m-1,ja:ja+n-1)$ to $beta$ on the
diagonal and $alpha$ on the off-diagonals. p?lase2 requires that only the dimension of the matrix operand is
distributed.

Input Parameters

$uplo$ (global)
Specifies the part of the distributed matrix sub( $A$ ) to be set:
= 'U': Upper triangular part is set; the strictly lower triangular part of
sub( $A$ ) is not changed;
= 'L': Lower triangular part is set; the strictly upper triangular part of
sub( $A$ ) is not changed;
Otherwise: All of the matrix sub( $A$ ) is set.

$m$ (global)
The number of rows to be operated on i.e the number of rows of the
distributed submatrix sub( $A$ ). $m >= 0$.

$n$ (global)
The number of columns to be operated on i.e the number of columns of the distributed submatrix sub( A ). \( n \geq 0 \).

\textbf{alpha} (global)
The constant to which the off-diagonal elements are to be set.

\textbf{beta} (global)
The constant to which the diagonal elements are to be set.

\textbf{ia} (global)
The row index in the global array \( a \) indicating the first row of sub( A ).

\textbf{ja} (global)
The column index in the global array \( a \) indicating the first column of sub( A ).

\textbf{desca} (global and local)
Array of size \( dlen_\_ \).
The array descriptor for the distributed matrix \( A \).

**Output Parameters**

\textbf{a} (local)
Pointer into the local memory to an array of size \( lld_a \* \text{LOCc(ja+}n-1) \).
This array contains the local pieces of the distributed matrix sub( A ) to be set.

On exit, the leading \( m \)-by-\( n \) submatrix sub( A ) is set as follows:
if \( uplo = 'U' \), \( A(ia+i-1,ja+j-1) = alpha, 1\leq i\leq j-1, 1\leq j\leq n \),
if \( uplo = 'L' \), \( A(ia+i-1,ja+j-1) = alpha, j+1\leq i\leq m, 1\leq j\leq n \),
otherwise, \( A(ia+i-1,ja+j-1) = alpha, 1\leq i\leq m, 1\leq j\leq n, ia+i \neq ja+j \),
and, for all \( uplo \), \( A(ia+i-1,ja+j-1) = beta, 1\leq i\leq \min(m,n) \).

\textbf{p?laset}
Initializes the offdiagonal elements of a matrix to \textit{alpha} and the diagonal elements to \textit{beta}.

**Syntax**

```c
void pslaset (char *uplo, MKL_INT *m, MKL_INT *n, float *alpha, float *beta, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca );
void pdlaset (char *uplo, MKL_INT *m, MKL_INT *n, double *alpha, double *beta, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca );
void pclaset (char *uplo, MKL_INT *m, MKL_INT *n, MKL_Complex8 *alpha, MKL_Complex8 *beta, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca );
void pzlaset (char *uplo, MKL_INT *m, MKL_INT *n, MKL_Complex16 *alpha, MKL_Complex16 *beta, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca );
```
Include Files

- mkl_scalapack.h

Description

The `p?laset` function initializes an \( m \)-by-\( n \) distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \) to \( beta \) on the diagonal and \( alpha \) on the offdiagonals.

Input Parameters

\( \text{uplo} \) (global)

Specifies the part of the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \) to be set:

- if \( \text{uplo} = 'U' \): upper triangular part; the strictly lower triangular part of \( A(ia:ia+m-1, ja:ja+n-1) \) is not changed;
- if \( \text{uplo} = 'L' \): lower triangular part; the strictly upper triangular part of \( A(ia:ia+m-1, ja:ja+n-1) \) is not changed.

Otherwise: all of the matrix \( A(ia:ia+m-1, ja:ja+n-1) \) is set.

\( m \) (global)

The number of rows in the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \). \((m \geq 0)\).

\( n \) (global)

The number of columns in the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \). \((n \geq 0)\).

\( \alpha \) (global).

The constant to which the offdiagonal elements are to be set.

\( \beta \) (global).

The constant to which the diagonal elements are to be set.

Output Parameters

\( a \) (local).

Pointer into the local memory to an array of size \( \text{lld}_{a} \times \text{LOCc}(ja+n-1) \).

This array contains the local pieces of the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \) to be set. On exit, the leading \( m \)-by-\( n \) matrix \( A(ia:ia+m-1, ja:ja+n-1) \) is set as follows:

- if \( \text{uplo} = 'U' \), \( A(ia+i-1, ja+j-1) = \alpha, 1 \leq i \leq j-1, 1 \leq j \leq n \),
- if \( \text{uplo} = 'L' \), \( A(ia+i-1, ja+j-1) = \alpha, j+1 \leq i \leq m, 1 \leq j \leq n \),

otherwise, \( A(ia+i-1, ja+j-1) = \alpha, 1 \leq i \leq m, 1 \leq j \leq n, ia+i \neq ja+j \), and, for all \( \text{uplo}, A(ia+i-1, ja+i-1) = \beta, 1 \leq i \leq \min(m,n) \).

\( \text{ia}, \text{ja} \) (global)

The column and row indices in the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \) indicating the first row and column of the matrix \( A(ia:ia+m-1, ja:ja+n-1) \), respectively.

\( \text{desca} \) (global and local)

Array of size \( \text{dlen}_\_ \). The array descriptor for the distributed matrix \( A(ia:ia+m-1, ja:ja+n-1) \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.
p?lasmsub
Looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero.

Syntax

```c
void pslasmsub (const float *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *k, const float *smlnum, float *buf, const MKL_INT *lwork);
void pdlasmsub (const double *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *k, const double *smlnum, double *buf, const MKL_INT *lwork);
void pclasmsub (const MKL_Complex8 *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *k, const float *smlnum, MKL_Complex8 *buf, const MKL_INT *lwork);
void pzlasmsub (const MKL_Complex16 *a, const MKL_INT *desca, const MKL_INT *i, const MKL_INT *l, MKL_INT *k, const double *smlnum, MKL_Complex16 *buf, const MKL_INT *lwork);
```

Include Files

- mkl_scalapack.h

Description

The p?lasmsub function looks for a small subdiagonal element from the bottom of the matrix that it can safely set to zero. This function performs a global maximum and must be called by all processes.

Input Parameters

- **a**  
  (local)  
  Array of size lld_a*LOCc(n_a).  
  On entry, the Hessenberg matrix whose tridiagonal part is being scanned. Unchanged on exit.

- **desca**  
  (global and local)  
  Array of size dlen_. The array descriptor for the distributed matrix A.

- **i**  
  (global)  
  The global location of the bottom of the unreduced submatrix of A. Unchanged on exit.

- **l**  
  (global)  
  The global location of the top of the unreduced submatrix of A. Unchanged on exit.

- **smlnum**  
  (global)  
  On entry, a "small number" for the given matrix. Unchanged on exit. The machine-dependent constants for the stopping criterion.

- **lwork**  
  (local)  
  This must be at least 2*ceil(ceil((i-l)/mb_a )/ lcm(nprow,npcol)). Here lcm is least common multiple and nprowxnpcol is the logical grid size.
Output Parameters

\( k \) (global)

On exit, this yields the bottom portion of the unreduced submatrix. This will satisfy: \( l \leq k \leq i-1 \).

\( buf \) (local)

Array of size \( lwork \).

Application Notes

This routine parallelizes the code from \( ?lahqr \) that looks for a single small subdiagonal element.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\( p?lasrt \)

Sorts the numbers in an array and the corresponding vectors in increasing order.

Syntax

```c
void pslasrt (const char* id, const MKL_INT* n, float* d, const float* q, const MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, float* work, const MKL_INT* lwork, MKL_INT* iwork, MKL_INT* info);
void pdlasrt (const char* id, const MKL_INT* n, double* d, const double* q, const MKL_INT* iq, const MKL_INT* jq, const MKL_INT* descq, double* work, const MKL_INT* lwork, MKL_INT* iwork, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

\( p?lasrt \) sorts the numbers in \( d \) and the corresponding vectors in \( q \) in increasing order.

Input Parameters

\( id \) (global)

= 'I': sort \( d \) in increasing order;

= 'D': sort \( d \) in decreasing order. (NOT IMPLEMENTED YET)

\( n \) (global)

The number of columns to be operated on i.e the number of columns of the distributed submatrix \( \text{sub}(Q) \). \( n \geq 0 \).

\( d \) (global)

Array, size \( (n) \)

\( q \) (local)

Pointer into the local memory to an array of size \( lld_q*LOCc(jq+n-1) \).
This array contains the local pieces of the distributed matrix \( A \) to be copied from.

\( \text{iq} \) (global)
The row index in the global array \( A \) indicating the first row of sub( \( Q \) ).

\( \text{jq} \) (global)
The column index in the global array \( A \) indicating the first column of sub( \( Q \) ).

\( \text{descq} \) (global and local)
Array of size \( \text{dien}_- \).
The array descriptor for the distributed matrix \( A \).

\( \text{work} \) (local)
Array, size (\( lwork \))

\( \text{lwork} \) (local)
The size of the array \( \text{work} \).
\[
\text{lwork} = \text{MAX}( n, \text{NP} \times (\text{NB} + \text{NQ}) ), \quad \text{where} \quad \text{NP} = \text{numroc}( n, \text{NB}, \text{MYROW}, \text{IAROW}, \text{NPROW} ), \quad \text{NQ} = \text{numroc}( n, \text{NB}, \text{MYCOL}, \text{DESCQ}(\text{csrc}_-), \text{NPCOL} ).
\]
\( \text{numroc} \) is a ScaLAPACK tool function.

\( \text{iwork} \) (local)
Array, size (\( liwork \))

\( \text{liwork} \) (local)
The size of the array \( \text{iwork} \).
\[
\text{liwork} = n + 2 \times \text{NB} + 2 \times \text{NPCOL}
\]

**Output Parameters**

\( d \)
On exit, the numbers in \( d \) are sorted in increasing order.

\( \text{info} \) (global)
\[
\text{info} = 0: \text{successful exit}
\]
\[
\text{info} < 0: \text{If the } i\text{-th argument is an array and the } j\text{-th entry had an illegal value, then } \text{info} = -((i \times 100 + j)), \text{if the } i\text{-th argument is a scalar and had an illegal value, then } \text{info} = -i.
\]

**plassq**
*Updates a sum of squares represented in scaled form.*

**Syntax**

void pslassq (MKL_INT *n, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, float *scale, float *sumsq);
void pdlassq (MKL_INT *n, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, double *scale, double *sumsq);
void pclassq (MKL_INT *n, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, float *scale, float *sumsq);
void pzlassq (MKL_INT *n, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx, double *scale, double *sumsq);

Include Files
• mkl_scalapack.h

Description
The plassq function returns the values scl and smsq such that
scl^2 * smsq = x_1^2 + ... + x_n^2 + scale^2 * sumsq,
where
x_i = sub(X) = \text{sub}(X) = X[ix + (jx-1)*m_x + (i-1)*incx] for pslassq/pdlassq,
x_i = sub(X) = \text{abs}(X) = \text{abs}(X[ix + (jx-1)*m_x + (i-1)*incx]) for pclassq/pzlassq.

For real functions pslassq/pdlassq the value of sumsq is assumed to be non-negative and scl returns the value
scl = max(scale, abs(x_i)).

For complex functions pclassq/pzlassq the value of sumsq is assumed to be at least unity and the value of ssq will then satisfy
1.0 \leq ssq \leq sumsq + 2n

Value scale is assumed to be non-negative and scl returns the value
\textbf{scl = max} (scale, abs(\text{real}(x_i)), abs(\text{imag}(x_i)))

For all functions plassq values scale and sumsq must be supplied in scale and sumsq respectively, and scale and sumsq are overwritten by scl and ssq respectively.

All functions plassq make only one pass through the vector sub(X).

Input Parameters

n (global)
The length of the distributed vector sub(x).

x
The array that stores the vector for which a scaled sum of squares is computed:
x[ix + (jx-1)*m_x + i*incx], 0 \leq i < n.

ix (global)
The row index in the global matrix X indicating the first row of sub(X).

jx (global)
The column index in the global matrix X indicating the first column of sub(X).

descx (global and local) array of size dlen_.
The array descriptor for the distributed matrix X.
incx

(global)
The global increment for the elements of $X$. Only two values of $incx$ are supported in this version, namely 1 and $m_x$. The argument $incx$ must not equal zero.

scale

(local).
On entry, the value $scale$ in the equation above.

sumsq

(local)
On entry, the value $sumsq$ in the equation above.

Output Parameters

scale

(local).
On exit, $scale$ is overwritten with $scl$, the scaling factor for the sum of squares.

sumsq

(local).
On exit, $sumsq$ is overwritten with the value $smsq$, the basic sum of squares from which $scl$ has been factored out.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?laswp

Performs a series of row interchanges on a general rectangular matrix.

Syntax

void pslaswp (char *direc, char *rowcol, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *k1, MKL_INT *k2, MKL_INT *ipiv);
void pdlaswp (char *direc, char *rowcol, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *k1, MKL_INT *k2, MKL_INT *ipiv);
void pclaswp (char *direc, char *rowcol, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *k1, MKL_INT *k2, MKL_INT *ipiv);
void pzlaswp (char *direc, char *rowcol, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *k1, MKL_INT *k2, MKL_INT *ipiv);

Include Files

• mkl_scalapack.h

Description

The p?laswp function performs a series of row or column interchanges on the distributed matrix $sub(A)=A(ia:ia+n-1, ja:ja+n-1)$. One interchange is initiated for each of rows or columns $k1$ through $k2$ of $sub(A)$. This function assumes that the pivoting information has already been broadcast along the process row or column. Also note that this function will only work for $k1$-$k2$ being in the same $mb$ (or $nb$) block. If you want to pivot a full matrix, use p?lapiv.
Input Parameters

direc (global)
Specifies in which order the permutation is applied:
= 'F' - forward,
= 'B' - backward.

rowcol (global)
Specifies if the rows or columns are permuted:
= 'R' - rows,
= 'C' - columns.

n (global)
If rowcol='R', the length of the rows of the distributed matrix A(*, ja:ja+n-1) to be permuted;
If rowcol='C', the length of the columns of the distributed matrix A(ia:ia +n-1 , *) to be permuted;

a (local)
Pointer into the local memory to an array of size lld_a * LOCC(n_a). On entry, this array contains the local pieces of the distributed matrix to which the row/columns interchanges will be applied.

ia (global)
The row index in the global matrix A indicating the first row of sub(A).

ja (global)
The column index in the global matrix A indicating the first column of sub(A).

desca (global and local) array of size dlen_.
The array descriptor for the distributed matrix A.

k1 (global)
The first element of ipiv for which a row or column interchange will be done.

k2 (global)
The last element of ipiv for which a row or column interchange will be done.

ipiv (local)
Array of size LOCr(m_a)+mb_a for row pivoting and LOCr(n_a)+nb_a for column pivoting. This array is tied to the matrix A, ipiv[k]=l implies rows (or columns) k+1 and l are to be interchanged, k = 0, 1, ..., size (ipiv) -1.

Output Parameters

A (local)
On exit, the permuted distributed matrix.
p?latra
Computes the trace of a general square distributed matrix.

Syntax
float pslatra (MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);
double pdlatra (MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);
void pclatra (MKL_Complex8 *, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);
void pzlatra (MKL_Complex16 *, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);

Include Files
• mkl_scalapack.h

Description
This function computes the trace of an n-by-n distributed matrix sub(A) denoting $A(ia:ia+n-1, ja:ja+n-1)$. The result is left on every process of the grid.

Input Parameters
n (global)
The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A). $n \geq 0$.
a (local).
Pointer into the local memory to an array of size $lld_a \ast LOCc(ja+n-1)$ containing the local pieces of the distributed matrix, the trace of which is to be computed.
ia, ja (global) The row and column indices respectively in the global matrix A indicating the first row and the first column of the matrix sub(A), respectively.
desca (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix A.

Output Parameters
val
The value returned by the function.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?latrd**

Reduces the first nb rows and columns of a symmetric/Hermitian matrix A to real tridiagonal form by an orthogonal/unitary similarity transformation.

**Syntax**

```c
void pslatrd (char *uplo, MKL_INT *n, MKL_INT *nb, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, float *tau, float *w, MKL_INT *iw, MKL_INT *jw, MKL_INT *descw, float *work);
```

```c
void pdlatrd (char *uplo, MKL_INT *n, MKL_INT *nb, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, double *tau, double *w, MKL_INT *iw, MKL_INT *jw, MKL_INT *descw, double *work);
```

```c
void pclatrd (char *uplo, MKL_INT *n, MKL_INT *nb, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, MKL_Complex8 *tau, MKL_Complex8 *w, MKL_INT *iw, MKL_INT *jw, MKL_INT *descw, MKL_Complex8 *work);
```

```c
void pzlatrd (char *uplo, MKL_INT *n, MKL_INT *nb, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, MKL_Complex16 *tau, MKL_Complex16 *w, MKL_INT *iw, MKL_INT *jw, MKL_INT *descw, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?latrd function reduces nb rows and columns of a real symmetric or complex Hermitian matrix sub(A) = A(ia:ia+n-1, ja:ja+n-1) to symmetric/complex tridiagonal form by an orthogonal/unitary similarity transformation Q'*sub(A)*Q, and returns the matrices V and W, which are needed to apply the transformation to the unreduced part of sub(A).

If uplo = U, p?latrd reduces the last nb rows and columns of a matrix, of which the upper triangle is supplied;

if uplo = L, p?latrd reduces the first nb rows and columns of a matrix, of which the lower triangle is supplied.

This is an auxiliary function called by p?sytrd/p?hetrd.

**Input Parameters**

- **uplo**

  (global)

  Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix sub(A) is stored:

  = 'U': Upper triangular

  = L: Lower triangular.

- **n**

  (global)

  The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A). n ≥ 0.

- **nb**

  (global)

  The number of rows and columns to be reduced.
**a**
Pointer into the local memory to an array of size \( lld_a \ast LOCc(ja+n-1) \).

On entry, this array contains the local pieces of the symmetric/Hermitian distributed matrix sub(\( A \)).

If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of sub(\( A \)) contains the upper triangular part of the matrix, and its strictly lower triangular part is not referenced.

If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of sub(\( A \)) contains the lower triangular part of the matrix, and its strictly upper triangular part is not referenced.

**ia**
(global)
The row index in the global matrix \( A \) indicating the first row of sub(\( A \)).

**ja**
(global)
The column index in the global matrix \( A \) indicating the first column of sub(\( A \)).

**desca**
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

**iw**
(global)
The row index in the global matrix \( W \) indicating the first row of sub(\( W \)).

**jw**
(global)
The column index in the global matrix \( W \) indicating the first column of sub(\( W \)).

**descw**
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( W \).

**work**
(local)
Workspace array of size \( nb_a \).

**Output Parameters**

**a**
(local)
On exit, if \( uplo = 'U' \), the last \( nb \) columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(\( A \)); the elements above the diagonal with the array \( tau \) represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors;

if \( uplo = 'L' \), the first \( nb \) columns have been reduced to tridiagonal form, with the diagonal elements overwriting the diagonal elements of sub(\( A \)); the elements below the diagonal with the array \( tau \) represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors.

**d**
(local)
Array of size \( LOCc(ja+n-1) \).

The diagonal elements of the tridiagonal matrix \( T \): \( d[i] = A(i+1,i+1), i = 0, 1, ..., LOCc(ja+n-1)-1 \). \( d \) is tied to the distributed matrix \( A \).
The off-diagonal elements of the tridiagonal matrix $T$:

- $e[i] = A(i+1, i+2)$ if $uplo = 'U'$,
- $e[i] = A(i+2, i+1)$ if $uplo = 'L'$,

$i = 0, 1, ..., LOCc(ja+n-1) - 1$.

$e$ is tied to the distributed matrix $A$.

**Application Notes**

If $uplo = 'U'$, the matrix $Q$ is represented as a product of elementary reflectors

$$Q = H(n) \times H(n-1) \times ... \times H(n-nb+1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau \cdot v \cdot v'$$

where $\tau$ is a real/complex scalar, and $v$ is a real/complex vector with $v(i:n) = 0$ and $v(i-1) = 1$; $v(1:i-1)$ is stored on exit in $A(ia:ia-i+1, ja+i)$, and $\tau$ in $tau[ja+i-2]$.

If $uplo = 'L'$, the matrix $Q$ is represented as a product of elementary reflectors

$$Q = H(1) \times H(2) \times ... \times H(nb)$$

Each $H(i)$ has the form

$$H(i) = I - \tau \cdot v \cdot v'$$

where $\tau$ is a real/complex scalar, and $v$ is a real/complex vector with $v(1:i) = 0$ and $v(i+1) = 1$; $v(i+2: n)$ is stored on exit in $A(ia+i+1: ia+n-1, ja+i-1)$, and $\tau$ in $tau[ja+i-2]$.

The elements of the vectors $v$ together form the $n$-by-$nb$ matrix $V$ which is needed, with $W$, to apply the transformation to the unreduced part of the matrix, using a symmetric/Hermitian rank-2k update of the form:

$\mathsf{sub}(A) := \mathsf{sub}(A) - vw' - wv'$.

The contents of $a$ on exit are illustrated by the following examples with

$n = 5$ and $nb = 2$: 

$$
\begin{array}{cccc}
\vdots & \vdots & \vdots & \vdots \\

\tau_1 & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\tau_2 & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\tau_3 & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\tau_4 & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\tau_5 & \vdots & \vdots & \vdots \\
\end{array}
$$
where \( d \) denotes a diagonal element of the reduced matrix, \( a \) denotes an element of the original matrix that is unchanged, and \( v_i \) denotes an element of the vector defining \( H(i) \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

### p?latrs

Solves a triangular system of equations with the scale factor set to prevent overflow.

**Syntax**

```c
void pslatrs (char *uplo, char *trans, char *diag, char *normin, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *scale, float *cnorm, float *work);

void pdlatrs (char *uplo, char *trans, char *diag, char *normin, MKL_INT *n, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *scale, double *cnorm, double *work);

void pclatrs (char *uplo, char *trans, char *diag, char *normin, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, float *scale, float *cnorm, MKL_Complex8 *work);

void pzlatrs (char *uplo, char *trans, char *diag, char *normin, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *x, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, double *scale, double *cnorm, MKL_Complex16 *work);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?latrs function solves a triangular system of equations \( Ax = sb \), \( A^T x = sb \) or \( A^H x = sb \), where \( s \) is a scale factor set to prevent overflow. The description of the function will be extended in the future releases.

**Input Parameters**

- **uplo**

  Specifies whether the matrix \( A \) is upper or lower triangular.
= 'U': Upper triangular
= 'L': Lower triangular

\textit{trans}

Specifies the operation applied to $Ax$.
= 'N': Solve $Ax = s*b$ (no transpose)
= 'T': Solve $A^\top x = s*b$ (transpose)
= 'C': Solve $A^H x = s*b$ (conjugate transpose),

where $s$ - is a scale factor

\textit{diag}

Specifies whether or not the matrix $A$ is unit triangular.
= 'N': Non-unit triangular
= 'U': Unit triangular

\textit{normin}

Specifies whether $cnorm$ has been set or not.
= 'Y': $cnorm$ contains the column norms on entry;
= 'N': $cnorm$ is not set on entry. On exit, the norms will be computed and stored in $cnorm$.

\textbf{n}

The order of the matrix $A$. $n \geq 0$

\textbf{a}

Array of size $lda* n$. Contains the triangular matrix $A$.

If $\text{uplo} = \text{U}$, the leading $n$-by-$n$ upper triangular part of the array $a$ contains the upper triangular matrix, and the strictly lower triangular part of $a$ is not referenced.

If $\text{uplo} = \text{L}$, the leading $n$-by-$n$ lower triangular part of the array $a$ contains the lower triangular matrix, and the strictly upper triangular part of $a$ is not referenced.

If $\text{diag} = \text{U}$, the diagonal elements of $a$ are also not referenced and are assumed to be 1.

\textbf{ia, ja}

(global) The row and column indices in the global matrix $A$ indicating the first row and the first column of the submatrix $A$, respectively.

\textbf{desca}

(global and local) array of size $dlen_\text{a}$. The array descriptor for the distributed matrix $A$.

\textbf{x}

Array of size $n$. On entry, the right hand side $b$ of the triangular system.

\textbf{ix}

(global).The row index in the global matrix $X$ indicating the first row of sub($X$).

\textbf{jx}

(global)

The column index in the global matrix $X$ indicating the first column of sub($X$).

\textbf{descx}

(global and local)

Array of size $dlen_\text{x}$. The array descriptor for the distributed matrix $X$. 
cnorm

Array of size n. If normin = 'Y', cnorm is an input argument and cnorm[j]
contains the norm of the off-diagonal part of the (j+1)-th column of the
matrix A, j=0, 1, ..., n-1. If trans = 'N', cnorm[j] must be greater than or
equal to the infinity-norm, and if trans = 'T' or 'C', cnorm[j] must be
greater than or equal to the 1-norm.

work

(local).
Temporary workspace.

Output Parameters

x

On exit, x is overwritten by the solution vector x.

scale

Array of size lda* n. The scaling factor s for the triangular system as
described above.

If scale = 0, the matrix A is singular or badly scaled, and the vector x is
an exact or approximate solution to Ax = 0.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

p?latrz

Reduces an upper trapezoidal matrix to upper
triangular form by means of orthogonal/unitary
transformations.

Syntax

void pslatrz (MKL_INT *m, MKL_INT *n, MKL_INT *l, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work);
void pdlatrz (MKL_INT *m, MKL_INT *n, MKL_INT *l, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work);
void pclatrz (MKL_INT *m, MKL_INT *n, MKL_INT *l, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work);
void pzlatrz (MKL_INT *m, MKL_INT *n, MKL_INT *l, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work);

Include Files

• mkl_scalapack.h

Description

The p?latrz function reduces the m-by-n (m ≤ n) real/complex upper trapezoidal matrix sub(A) =
[A(ia:ia+m-1, ja:ja+m-1)A(ia:ia+m-1, ja+n-l:ja+n-l)] to upper triangular form by means of
orthogonal/unitary transformations.

The upper trapezoidal matrix sub(A) is factored as
sub(A) = ( R 0 )*Z,

where Z is an n-by-n orthogonal/unitary matrix and R is an m-by-m upper triangular matrix.
Input Parameters

\( m \) (global)
The number of rows in the distributed matrix \( \text{sub}(A) \). \( m \geq 0 \).

\( n \) (global)
The number of columns in the distributed matrix \( \text{sub}(A) \). \( n \geq 0 \).

\( l \) (global)
The number of columns of the distributed matrix \( \text{sub}(A) \) containing the meaningful part of the Householder reflectors. \( l > 0 \).

\( a \) (local)
Pointer into the local memory to an array of size \( \text{lld}_a \ast \text{LOCc}(ja+n-1) \). On entry, the local pieces of the \( m \)-by-\( n \) distributed matrix \( \text{sub}(A) \), which is to be factored.

\( ia \) (global)
The row index in the global matrix \( A \) indicating the first row of \( \text{sub}(A) \).

\( ja \) (global)
The column index in the global matrix \( A \) indicating the first column of \( \text{sub}(A) \).

\( \text{desca} \) (global and local) array of size \( \text{dlen}_a \).
The array descriptor for the distributed matrix \( A \).

\( \text{work} \) (local)
Workspace array of size \( \text{lwork} \).
\( \text{lwork} \geq \text{nq0} + \max(1, \text{mp0}) \), where
\( \text{iroff} = \text{mod}(ia-1, \text{mb}_a) \),
\( \text{icoff} = \text{mod}(ja-1, \text{nb}_a) \),
\( \text{iarow} = \text{indxg2p}(ia, \text{mb}_a, \text{myrow}, \text{rsr}_c_a, \text{npro}_w) \),
\( \text{iacol} = \text{indxg2p}(ja, \text{nb}_a, \text{mycol}, \text{csr}_c_a, \text{npcol}) \),
\( \text{mp0} = \text{numroc}(m+\text{iroff}, \text{mb}_a, \text{myrow}, \text{iarow}, \text{npro}_w) \),
\( \text{nq0} = \text{numroc}(n+\text{icoff}, \text{nb}_a, \text{mycol}, \text{iacol}, \text{npcol}) \),
\( \text{numroc}, \text{indxg2p}, \text{and numroc} \) are ScaLAPACK tool functions; \( \text{myrow}, \text{mycol}, \text{npro}_w, \text{andnpcol} \) can be determined by calling the function \( \text{blacs_gridinfo} \).

Output Parameters

\( a \)
On exit, the leading \( m \)-by-\( m \) upper triangular part of \( \text{sub}(A) \) contains the upper triangular matrix \( R \), and elements \( n-l+1 \) to \( n \) of the first \( m \) rows of \( \text{sub}(A) \), with the array \( \text{tau} \), represent the orthogonal/unitary matrix \( Z \) as a product of \( m \) elementary reflectors.

\( \text{tau} \) (local)
Array of size $\text{LOCr}(ja+m-1)$. This array contains the scalar factors of the elementary reflectors. $\tau$ is tied to the distributed matrix $A$.

**Application Notes**

The factorization is obtained by Householder's method. The $k$-th transformation matrix, $Z(k)$, which is used (or, in case of complex functions, whose conjugate transpose is used) to introduce zeros into the $(m - k + 1)$-th row of $\text{sub}(A)$, is given in the form

$$Z(k) = \begin{bmatrix} I & 0 \\ 0 & T(k) \end{bmatrix},$$

where

$$T(k) = I - \tau u(k) u(k)^\dagger,$$ $$u(k) = \begin{bmatrix} 1 \\ 0 \\ z(k) \end{bmatrix}$$

$\tau$ is a scalar and $z(k)$ is an $(n-m)$-element vector. $\tau$ and $z(k)$ are chosen to annihilate the elements of the $k$-th row of $\text{sub}(A)$. The scalar $\tau$ is returned in the $k$-th element of $\tau$, indexed $k-1$, and the vector $u(k)$ in the $k$-th row of $\text{sub}(A)$, such that the elements of $z(k)$ are in $A(k, m+1), \ldots, A(k, n)$. The elements of $R$ are returned in the upper triangular part of $\text{sub}(A)$.

$Z$ is given by

$$Z = Z(1)Z(2)\ldots Z(m).$$

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?lauu2**

*Computes the product $U^*U'$ or $L'*L$, where $U$ and $L$ are upper or lower triangular matrices (local unblocked algorithm).*

**Syntax**

```c
void pslauu2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdlauu2 (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pclauu2 (char *uplo , MKL_INT *n , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pzlauu2 (char *uplo , MKL_INT *n , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?lauu2 function computes the product $U^*U'$ or $L'*L$, where the triangular factor $U$ or $L$ is stored in the upper or lower triangular part of the distributed matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $\text{uplo} = 'U'$ or '$u'$, then the upper triangle of the result is stored, overwriting the factor $U$ in $\text{sub}(A)$.
If \( uplo = 'L' \) or \( 'l' \), then the lower triangle of the result is stored, overwriting the factor \( L \) in \( \text{sub}(A) \).

This is the unblocked form of the algorithm, calling BLAS Level 2 Routines. No communication is performed by this function, the matrix to operate on should be strictly local to one process.

**Input Parameters**

- \( \text{uplo} \) (global)
  
  Specifies whether the triangular factor stored in the matrix \( \text{sub}(A) \) is upper or lower triangular:
  
  \( = 'U' \): upper triangular

  \( = 'L' \): lower triangular.

- \( n \) (global)
  
  The number of rows and columns to be operated on, that is, the order of the triangular factor \( U \) or \( L \). \( n \geq 0 \).

- \( a \) (local)
  
  Pointer into the local memory to an array of size \( \text{lld}_a * \text{LOCc}(ja+n-1) \). On entry, the local pieces of the triangular factor \( U \) or \( L \).

- \( \text{ia} \) (global)
  
  The row index in the global matrix \( A \) indicating the first row of \( \text{sub}(A) \).

- \( \text{ja} \) (global)
  
  The column index in the global matrix \( A \) indicating the first column of \( \text{sub}(A) \).

- \( \text{desca} \) (global and local) array of size \( \text{dlen}_a \). The array descriptor for the distributed matrix \( A \).

**Output Parameters**

- \( a \) (local)
  
  On exit, if \( \text{uplo} = 'U' \), the upper triangle of the distributed matrix \( \text{sub}(A) \) is overwritten with the upper triangle of the product \( U^*U \); if \( \text{uplo} = 'L' \), the lower triangle of \( \text{sub}(A) \) is overwritten with the lower triangle of the product \( L^*L \).

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?lauum**

*Computes the product \( U^*U \) or \( L^*L \), where \( U \) and \( L \) are upper or lower triangular matrices.*

**Syntax**

```c
void pslauum (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
void pdlauum (char *uplo , MKL_INT *n , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca );
```
void pclauum (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);

void pzlauum (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca);

Include Files
• mkl_scalapack.h

Description
The p?lauum function computes the product $U^*U'$ or $L'^*L$, where the triangular factor $U$ or $L$ is stored in the upper or lower triangular part of the matrix $\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)$.

If $\text{uplo} = 'U'$ or 'u', then the upper triangle of the result is stored, overwriting the factor $U$ in $\text{sub}(A)$. If $\text{uplo} = 'L'$ or 'l', then the lower triangle of the result is stored, overwriting the factor $L$ in $\text{sub}(A)$.

This is the blocked form of the algorithm, calling Level 3 PBLAS.

Input Parameters

$\text{uplo}$ (global)
Specifies whether the triangular factor stored in the matrix $\text{sub}(A)$ is upper or lower triangular:

= 'U': upper triangular
= 'L': lower triangular.

$n$ (global)
The number of rows and columns to be operated on, that is, the order of the triangular factor $U$ or $L$. $n \geq 0$.

$a$ (local)
Pointer into the local memory to an array of size $\text{lld}_a \ast \text{LOCc}(ja+n-1)$. On entry, the local pieces of the triangular factor $U$ or $L$.

$\text{ia}$ (global)
The row index in the global matrix $A$ indicating the first row of $\text{sub}(A)$.

$\text{ja}$ (global)
The column index in the global matrix $A$ indicating the first column of $\text{sub}(A)$.

$\text{desca}$ (global and local) array of size $\text{dlen}_$. The array descriptor for the distributed matrix $A$.

Output Parameters

$a$ (local)
On exit, if $\text{uplo} = 'U'$, the upper triangle of the distributed matrix $\text{sub}(A)$ is overwritten with the upper triangle of the product $U^*U'$; if $\text{uplo} = 'L'$, the lower triangle of $\text{sub}(A)$ is overwritten with the lower triangle of the product $L'^*L$.  

Intel® Math Kernel Library Developer Reference
See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lawil
Forms the Wilkinson transform.

Syntax

void pslawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const float *a, const MKL_INT *desca, const float *h44, const float *h33, const float *h43h34, float *v);

void pdlawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const double *a, const MKL_INT *desca, const double *h44, const double *h33, const double *h43h34, double *v);

void pclawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const MKL_Complex8 *a, const MKL_INT *desca, const MKL_Complex8 *h44, const MKL_Complex8 *h33, const MKL_Complex8 *h43h34, MKL_Complex8 *v);

void pzlawil (const MKL_INT *ii, const MKL_INT *jj, const MKL_INT *m, const MKL_Complex16 *a, const MKL_INT *desca, const MKL_Complex16 *h44, const MKL_Complex16 *h33, const MKL_Complex16 *h43h34, MKL_Complex16 *v);

Include Files
• mkl_scalapack.h

Description
The p?lawil function gets the transform given by $h_{44}$, $h_{33}$, and $h_{43}h_{34}$ into $v$ starting at row $m$.

Input Parameters

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$ii$</td>
<td>(global)</td>
</tr>
<tr>
<td>Number of the process row which owns the matrix element $A(m+2, m+2)$.</td>
<td></td>
</tr>
<tr>
<td>$jj$</td>
<td>(global)</td>
</tr>
<tr>
<td>Number of the process column which owns the matrix element $A(m+2, m+2)$.</td>
<td></td>
</tr>
<tr>
<td>$m$</td>
<td>(global)</td>
</tr>
<tr>
<td>On entry, the location from where the transform starts (row $m$). Unchanged on exit.</td>
<td></td>
</tr>
<tr>
<td>$a$</td>
<td>(local)</td>
</tr>
<tr>
<td>Array of size $lld_a*LOCc(n_a)$. On entry, the Hessenberg matrix. Unchanged on exit.</td>
<td></td>
</tr>
<tr>
<td>desca</td>
<td>(global and local)</td>
</tr>
<tr>
<td>Array of size $dlen_$. The array descriptor for the distributed matrix $A$. Unchanged on exit.</td>
<td></td>
</tr>
<tr>
<td>$h_{43}h_{34}$</td>
<td>(global)</td>
</tr>
<tr>
<td>These three values are for the double shift QR iteration. Unchanged on exit.</td>
<td></td>
</tr>
</tbody>
</table>
Output Parameters

\( v \)  
\( (\text{global}) \)  
Array of size 3 that contains the transform on output.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?org2l/p?ung2l**

Generates all or part of the orthogonal/unitary matrix \( Q \) from a QL factorization determined by \( p?geqlf \) (unblocked algorithm).

**Syntax**

```c
void psorg2l (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorg2l (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcung2l (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzung2l (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The \( p?org2l/p?ung2l \) function generates an \( m \)-by-\( n \) real/complex distributed matrix \( Q \) denoting \( A(ia:ia+m-1, ja:ja+n-1) \) with orthonormal columns, which is defined as the last \( n \) columns of a product of \( k \) elementary reflectors of order \( m \):

\[
Q = H(k)*...*H(2)*H(1)
\]

as returned by \( p?geqlf \).

**Input Parameters**

\( m \)  
\( (\text{global}) \)  
The number of rows in the distributed submatrix \( Q \). \( m \geq 0 \).

\( n \)  
\( (\text{global}) \)  
The number of columns in the distributed submatrix \( Q \). \( m \geq n \geq 0 \).

\( k \)  
\( (\text{global}) \)  
The number of elementary reflectors whose product defines the matrix \( Q \). \( n \geq k \geq 0 \).

\( a \)  
Pointer into the local memory to an array of size \( lld_a \times LOCC(ja+n-1) \).
On entry, the $j$-th column of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(j)$, $ja+n-k \leq j \leq ja+n-k$, as returned by p?geqlf in the $k$ columns of its distributed matrix argument $A(ia:,ja+n-k:ja+n-1)$.

$ia$ (global)
The row index in the global matrix $A$ indicating the first row of sub($A$).

$ja$ (global)
The column index in the global matrix $A$ indicating the first column of sub($A$).

desca (global and local) array of size $dlen_\$. The array descriptor for the distributed matrix $A$.

tau (local)
Array of size $\text{LOCc}(ja+n-1)$.
$\tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \ldots, \text{LOCc}(ja+n-1)-1$, as returned by p?geqlf.

work (local)
Workspace array of size $lwork$.

lwork (local or global)
The size of the array work.
$lwork$ is local input and must be at least $lwork \geq mpa_0 + \max(1, nqa_0)$, where

\begin{align*}
ipa & = \text{mod}(ia-1, mb_a), \\
ic & = \text{mod}(ja-1, nb_a), \\
\text{irow} & = \text{indxg2p}(ia, mb_a, myrow, rsrch_a, nprow), \\
\text{icol} & = \text{indxg2p}(ja, nb_a, mycol, csrch_a, npcol), \\
mpa_0 & = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow), \\
nqa_0 & = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol).
\end{align*}

indxg2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

Output Parameters

$a$
On exit, this array contains the local pieces of the $m$-by-$n$ distributed matrix $Q$.

work
On exit, $\text{work}[0]$ returns the minimal and optimal $lwork$.

info (local)
= 0: successful exit
< 0: if the i-th argument is an array and the j-th entry, indexed j-1, had an illegal value,
then info = -(i*100 +j),
if the i-th argument is a scalar and had an illegal value,
then info = -i.

See Also
Overview for details of ScalAPACK array descriptor structures and related notations.

p?org2r/p?ung2r
Generates all or part of the orthogonal/unitary matrix Q from a QR factorization determined by p?geqrf (unblocked algorithm).

Syntax
void psorg2r (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorg2r (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcung2r (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzung2r (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The p?org2r/p?ung2r function generates an m-by-n real/complex matrix Q denoting A(ia:ia+m-1, ja:ja+n-1) with orthonormal columns, which is defined as the first n columns of a product of k elementary reflectors of order m:
Q = H(1)*H(2)*...*H(k)
as returned by p?geqrf.

Input Parameters

m (global)
The number of rows in the distributed submatrix Q. m≥ 0.

n (global)
The number of columns in the distributed submatrix Q. m≥ n ≥ 0.
The number of elementary reflectors whose product defines the matrix $Q$. $n \geq k \geq 0$.

$a$

Pointer into the local memory to an array of size $lld_a \cdot \text{LOCc}(ja+n-1)$.

On entry, the $j$-th column of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by $p?geqrf$ in the $k$ columns of its distributed matrix argument $A(ia:*,ja:ja+k-1)$.

$ia$

(global)

The row index in the global matrix $A$ indicating the first row of sub($A$).

$ja$

(global)

The column index in the global matrix $A$ indicating the first column of sub($A$).

$desca$

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$tau$

(local)

Array of size $\text{LOCc}(ja+k-1)$.

$tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \ldots, \text{LOCc}(ja+k-1)-1$, as returned by $p?geqrf$. This array is tied to the distributed matrix $A$.

$work$

(local)

Workspace array of size $lwork$.

$lwork$

(local or global)

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq mpa0 + \max(1, nqa0)$, where

- $iroffa = \text{mod}(ia-1, mb_a)$, $icoffa = \text{mod}(ja-1, nb_a)$,
- $iarow = \text{indxg2p}(ia, mb_a, myrow, rsra_a, nprow)$,
- $iacol = \text{indxg2p}(ja, nb_a, mycol, rsca_a, npcol)$,
- $mpa0 = \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow)$,
- $nqa0 = \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol)$.

$\text{indxg2p}$ and $\text{numroc}$ are ScaLAPACK tool functions; $myrow$, $mycol$, $nprow$, and $npcol$ can be determined by calling the function $\text{blacs_gridinfo}$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by $pxerbla$. 

---

*ScaLAPACK Routines* 4
Output Parameters

- **a**
  - On exit, this array contains the local pieces of the $m$-by-$n$ distributed matrix $Q$.

- **work**
  - On exit, `work[0]` returns the minimal and optimal `lwork`

- **info**
  - $= 0$: successful exit
  - $< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value,
  - then $info = - (i*100 + j)$,
  - if the $i$-th argument is a scalar and had an illegal value,
  - then $info = -i$.

See Also

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?orgl2/p?ungl2**
Generates all or part of the orthogonal/unitary matrix $Q$ from an LQ factorization determined by `p?gelqf` (unblocked algorithm).

### Syntax

```c
void psorgl2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorgl2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pcungl2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzungl2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `p?orgl2/p?ungl2` function generates a $m$-by-$n$ real/complex matrix $Q$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the first $m$ rows of a product of $k$ elementary reflectors of order $n$

- $Q = H(k)*...*H(2)*H(1)$ (for real flavors),
- $Q = (H(k))^H*...*(H(2))^H*(H(1))^H$ (for complex flavors) as returned by `p?gelqf`.

**Input Parameters**

- **m**
  - (global)
The number of rows in the distributed submatrix $Q. m \geq 0$.

$n$

The number of columns in the distributed submatrix $Q. n \geq m \geq 0$.

$k$

The number of elementary reflectors whose product defines the matrix $Q. m \geq k \geq 0$.

$a$

Pointer into the local memory to an array of size $lld_a \times \text{LOCc}(ja+n-1)$.

On entry, the $i$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(i), ia \leq i \leq ia+k-1$, as returned by $p?gelqf$ in the $k$ rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

$ia$

The row index in the global matrix $A$ indicating the first row of sub($A$).

$ja$

The column index in the global matrix $A$ indicating the first column of sub($A$).

desca

(global and local) array of size $dlen_. The array descriptor for the distributed matrix $A$.

tau

(local)

Array of size $\text{LOCr}(ja+k-1). \tau[j]$ contains the scalar factor of the elementary reflectors $H(j+1), j = 0, 1, ..., \text{LOCr}(ja+k-1)-1$, as returned by $p?gelqf$. This array is tied to the distributed matrix $A$.

$work$

(local)

Workspace array of size $lwork$.

$lwork$

(local or global)

The size of the array $work$.

$lwork$ is local input and must be at least $lwork \geq nqa0 + \max(1, mpa0)$, where

\[
\begin{align*}
  iroffa &= \text{mod}(ia-1, mb_a), \\
  icoffa &= \text{mod}(ja-1, nb_a), \\
  iarow &= \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow), \\
  iacol &= \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol), \\
  mpa0 &= \text{numroc}(m+iroffa, mb_a, myrow, iarow, nprow), \\
  nqa0 &= \text{numroc}(n+icoffa, nb_a, mycol, iacol, npcol).
\end{align*}
\]

$\text{indxg2p}$ and $\text{numroc}$ are ScaLAPACK tool functions; $myrow, mycol, nprow,$ and $npcol$ can be determined by calling the function $\text{blacs\_gridinfo}$. 

ScaLAPACK Routines
If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

**Output Parameters**

$a$

On exit, this array contains the local pieces of the $m$-by-$n$ distributed matrix $Q$.

$work$

On exit, $work[0]$ returns the minimal and optimal $lwork$.

$info$

(local)

- $0$: successful exit
- $< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*100 +j)$.
- if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?orgr2/p?ungr2**

Generates all or part of the orthogonal/unitary matrix $Q$ from an RQ factorization determined by $p?gerqf$ (unblocked algorithm).

**Syntax**

```c
void psorgr2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *work, MKL_INT *lwork, MKL_INT *info );
void pdorgr2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *work, MKL_INT *lwork, MKL_INT *info );
void pcungr2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info );
void pzungr2 (MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**

The $p?orgr2/p?ungr2$ function generates an $m$-by-$n$ real/complex matrix $Q$ denoting $A(ia:ia+m-1, ja:ja+n-1)$ with orthonormal rows, which is defined as the last $m$ rows of a product of $k$ elementary reflectors of order $n$:

$$Q = H(1)H(2)*...*H(k)$$

(for real flavors);
\[ Q = (H(1))^{H}*(H(2))^{H}...*(H(k))^{H} \text{ (for complex flavors) as returned by } \text{pgerqf}. \]

**Input Parameters**

\( m \) (global)

The number of rows in the distributed submatrix \( Q \). \( m \geq 0 \).

\( n \) (global)

The number of columns in the distributed submatrix \( Q \). \( n \geq m \geq 0 \).

\( k \) (global)

The number of elementary reflectors whose product defines the matrix \( Q \). \( m \geq k \geq 0 \).

\( a \)

Point into the local memory to an array of size \( lld_a \times LOCc(ja+n-1) \).

On entry, the \( i \)-th row of the matrix stored in \( a \) must contain the vector that defines the elementary reflector \( H(i) \), \( ia+m-k \leq i \leq ia+m-1 \), as returned by \text{pgerqf} in the \( k \) rows of its distributed matrix argument \( A(ia+m-k:ia+m-1, ja:*). \)

\( ia \) (global)

The row index in the global matrix \( A \) indicating the first row of \( \text{sub}(A) \).

\( ja \) (global)

The column index in the global matrix \( A \) indicating the first column of \( \text{sub}(A) \).

\( \text{desca} \) (global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( \text{tau} \) (local)

Array of size \( \text{LOCr}(ja+m-1) \). \( \text{tau}[j] \) contains the scalar factor of the elementary reflectors \( H(j+1), j = 0, 1, \ldots, \text{LOCr}(ja+m-1)-1 \), as returned by \text{pgerqf}. This array is tied to the distributed matrix \( A \).

\( \text{work} \) (local)

Workspace array of size \( lwork \).

\( lwork \) (local or global)

The size of the array \( \text{work} \).

\( lwork \) is local input and must be at least \( lwork \geq nqa0 + \max(1, mpa0) \), where \( iroff = \text{mod}(ia-1, mb_a) \), \( icoff = \text{mod}(ja-1, nb_a) \),

\( iarow = \text{indxg2p}(ia, mb_a, myrow, rsrc_a, nprow) \),

\( iacol = \text{indxg2p}(ja, nb_a, mycol, csrc_a, npcol) \),

\( mpa0 = \text{numroc}(m+iroff, mb_a, myrow, iarow, nprow) \),

\( nqa0 = \text{numroc}(n+icoff, nb_a, mycol, iacol, npcol) \).

\text{indxg2p} and \text{numroc} are \text{ScaLAPACK} tool functions; \( \text{myrow, mycol, nprow, and npcol} \) can be determined by calling the function \text{blacs_gridinfo}. 

---

**ScaLAPACK Routines**

4 1641
If \( lwork = -1 \), then \( lwork \) is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \texttt{pxerbla}.

**Output Parameters**

\( a \)

On exit, this array contains the local pieces of the \( m \)-by-\( n \) distributed matrix \( Q \).

\( \text{work} \)

On exit, \( \text{work}[0] \) returns the minimal and optimal \( lwork \).

\( \text{info} \)

(local)

\[ = 0: \text{successful exit} \]

\[ < 0: \text{if the} \ i\text{-th argument is an array and the} \ j\text{-th entry, indexed} \ j-1, \text{had an illegal value, then} \text{info} = - (i*100 + j) , \]

if the \( i\text{-th argument is a scalar and had an illegal value, then} \text{info} = -i. \]

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**\texttt{porm2l}/\texttt{punm2l}**

Multiples a general matrix by the orthogonal/unitary matrix from a QL factorization determined by \texttt{pgeqlf} (unblocked algorithm).

**Syntax**

```c
void psorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *tau , float *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descd , float *work , MKL_INT *lwork , MKL_INT *info );
void pdorm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , double *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , double *tau , double *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descd , double *work , MKL_INT *lwork , MKL_INT *info );
void pcunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex8 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *tau , MKL_Complex8 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descd , MKL_Complex8 *work , MKL_INT *lwork , MKL_INT *info );
void pzunm2l (char *side , char *trans , MKL_INT *m , MKL_INT *n , MKL_INT *k , MKL_Complex16 *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_Complex16 *tau , MKL_Complex16 *c , MKL_INT *ic , MKL_INT *jc , MKL_INT *descd , MKL_Complex16 *work , MKL_INT *lwork , MKL_INT *info );
```

**Include Files**

- \texttt{mkl_scalapack.h}

1642
Description
The function overwrites the general real/complex $m$-by-$n$ distributed matrix sub
\[
(C) = C(\text{ic:ic+m-1,jc:jc+n-1})
\]
with
\[
Q\text{*sub}(C) \text{ if } \text{side} = 'L' \text{ and } \text{trans} = 'N', \text{ or}
\]
\[
Q^T\text{*sub}(C) / Q^H\text{*sub}(C) \text{ if } \text{side} = 'L' \text{ and } \text{trans} = 'T' \text{ (for real flavors) or } \text{trans} = 'C' \text{ (for complex flavors)}, \text{ or}
\]
\[
\text{sub}(C)\text{*}Q \text{ if } \text{side} = 'R' \text{ and } \text{trans} = 'N', \text{ or}
\]
\[
\text{sub}(C)\text{*}Q^T / \text{sub}(C)\text{*}Q^H \text{ if } \text{side} = 'R' \text{ and } \text{trans} = 'T' \text{ (for real flavors) or } \text{trans} = 'C' \text{ (for complex flavors)},
\]
where $Q$ is a real orthogonal or complex unitary distributed matrix defined as the product of $k$ elementary reflectors
\[
Q = H(k)*...*H(2)*H(1)
\]
as returned by \texttt{pgeqlf}. $Q$ is of order $m$ if $\text{side} = 'L'$ and of order $n$ if $\text{side} = 'R'$.

Input Parameters

- \textit{side} (global)
  - $'L'$: apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the left,
  - $'R'$: apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the right.

- \textit{trans} (global)
  - $'N'$: apply $Q$ (no transpose)
  - $'T'$: apply $Q^T$ (transpose, for real flavors)
  - $'C'$: apply $Q^H$ (conjugate transpose, for complex flavors)

- \textit{m} (global)
The number of rows in the distributed matrix sub($C$). $m \geq 0$.

- \textit{n} (global)
The number of columns in the distributed matrix sub($C$). $n \geq 0$.

- \textit{k} (global)
The number of elementary reflectors whose product defines the matrix $Q$.
  If $\text{side} = 'L'$, $m \geq k \geq 0$;
  if $\text{side} = 'R'$, $n \geq k \geq 0$.

- \textit{a} (local)
  Pointer into the local memory to an array of size $\text{lld}_a \ast \text{LOCc}(ja+k-1)$.
  On entry, the $j$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(j)$, $ja \leq j \leq ja+k-1$, as returned by \texttt{pgeqlf} in the $k$ columns of its distributed matrix argument $A(ia:*,ja:ja+k-1)$. The argument $A(ia:*,ja:ja+k-1)$ is modified by the function but restored on exit.
  If $\text{side} = 'L'$, $\text{lld}_a \geq \max(1, \text{LOCr}(ia+m-1))$,
if side = 'R', lld_a ≥ max(1, LOCr(ia+n-1)).

ia (global)
The row index in the global matrix A indicating the first row of sub(A).

ja (global)
The column index in the global matrix A indicating the first column of sub(A).

desca (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

tau (local)
Array of size LOCC(ja+n-1). tau[j] contains the scalar factor of the elementary reflector H(j+1), j = 0, 1, ..., LOCC(ja+n-1)-1, as returned by pgeqlf. This array is tied to the distributed matrix A.

c (local)
Pointer into the local memory to an array of size lld_c * LOCC(jc+n-1). On entry, the local pieces of the distributed matrix sub (C).

ic (global)
The row index in the global matrix C indicating the first row of sub(C).

jc (global)
The column index in the global matrix C indicating the first column of sub(C).

descc (global and local) array of size dlen_. The array descriptor for the distributed matrix C.

work (local)
Workspace array of size lwork.

On exit, work(1) returns the minimal and optimal lwork.

lwork (local or global)
The size of the array work.

lwork is local input and must be at least

if side = 'L', lwork ≥ mpc0 + max(1, nqc0),

if side = 'R', lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n + icoffc, nb_a, 0, 0, npcol), nb_a, 0, 0, lcmq)),

where

lcmq = lcm/npcol,

lcm = iclm(nprow, npcol),

iwoffc = mod(ic-1, mb_c),

icoffc = mod(jc-1, nb_c),

icrow = indxg2p(ic, mb_c, myrow, rsr_c, nprow),

iccol = indxg2p(jc, nb_c, mycol, csr_c, npcol),
Output Parameters

c
On exit, c is overwritten by \( Q \cdot \text{sub}(C) \), or \( Q^T \cdot \text{sub}(C) \), or \( Q^H \cdot \text{sub}(C) \).

work
On exit, work[0] returns the minimal and optimal lwork.

info
(local)
= 0: successful exit
< 0: if the i-th argument is an array and the j-th entry, indexed j-1, had an illegal value,
then info = - (i*100 +j),
if the i-th argument is a scalar and had an illegal value,
then info = -i.

NOTE
The distributed submatrices \( A(ia:*, ja:* \) and \( C(ic:ic+m-1, jc:ic+n-1) \) must verify some alignment properties, namely the following expressions should be true:

If side = 'L', \( (mb_a == mb_c && iroff_a == iroff_c && iarow == icrow) \)
If side = 'R', \( (mb_a == nb_c && iroff_a == iroff_c) \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?orm2r/p?unm2r
Multiplies a general matrix by the orthogonal/unitary matrix from a QR factorization determined by
p?geqrf (unblocked algorithm).

Syntax

void psorm2r (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, float *work, MKL_INT *lwork, MKL_INT *info);
void pdorm2r (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descb, double *work, MKL_INT *lwork, MKL_INT *info);
void pcunm2r (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzunm2r (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The p?orm2r/p?unm2r function overwrites the general real/complex m-by-n distributed matrix sub
\( C = C(\text{ic} : \text{ic}+m-1, \text{jc} : \text{jc}+n-1) \) with
\[ Q^* \cdot \text{sub}(C) \] if side = 'L' and trans = 'N', or
\[ Q^T \cdot \text{sub}(C) / \; Q^H \cdot \text{sub}(C) \] if side = 'L' and trans = 'T' (for real flavors) or trans = 'C' (for complex flavors), or
\[ \text{sub}(C) \cdot Q \] if side = 'R' and trans = 'N', or
\[ \text{sub}(C) \cdot Q^T / \; \text{sub}(C) \cdot Q^H \] if side = 'R' and trans = 'T' (for real flavors) or trans = 'C' (for complex flavors).

where \( Q \) is a real orthogonal or complex unitary matrix defined as the product of \( k \) elementary reflectors
\[ Q = H(k) \cdots H(2) \cdot H(1) \] as returned by p?geqrf. \( Q \) is of order \( m \) if side = 'L' and of order \( n \) if side = 'R'.

Input Parameters

side

(global)
= 'L': apply \( Q \) or \( Q^T \) for real flavors (\( Q^H \) for complex flavors) from the left,
= 'R': apply \( Q \) or \( Q^T \) for real flavors (\( Q^H \) for complex flavors) from the right.

trans

(global)
= 'N': apply \( Q \) (no transpose)
= 'T': apply \( Q^T \) (transpose, for real flavors)
= 'C': apply \( Q^H \) (conjugate transpose, for complex flavors)

m

(global)
The number of rows in the distributed matrix sub(C). \( m \geq 0 \).

n

(global)
The number of columns in the distributed matrix sub(C). \( n \geq 0 \).

k

(global)
The number of elementary reflectors whose product defines the matrix \( Q \).
If \( side = 'L', m \geq k \geq 0 \);
if side = 'R', n ≥ k ≥ 0.

a
(local)
Pointer into the local memory to an array of size lld_a * LOCc(ja+k-1).

On entry, the j-th column of the matrix stored in a must contain the vector
that defines the elementary reflector H(j), ja ≤ j ≤ ja + k - 1, as returned by
pgeqrf in the k columns of its distributed matrix argument A(ia:*, ja:ja + k-1).
The argument A(ia:*, ja:ja+k-1) is modified by the function but
restored on exit.

If side = 'L', lld_a ≥ max(1, LOCr(ia+m-1)),
If side = 'R', lld_a ≥ max(1, LOCr(ia+n-1)).

ia
(global)
The row index in the global matrix A indicating the first row of sub(A).

ja
(global)
The column index in the global matrix A indicating the first column of
sub(A).

desca
(global and local) array of size dlen_. The array descriptor for the
distributed matrix A.

tau
(local)
Array of size LOCc(ja+k-1). tau[j] contains the scalar factor of the
elementary reflector H(j+1), j = 0, 1, ..., LOCc(ja+k-1)-1, as returned by
pgeqrf. This array is tied to the distributed matrix A.

c
(local)
Pointer into the local memory to an array of size lld_c * LOCc(jc+n-1).

On entry, the local pieces of the distributed matrix sub (C).

ic
(global)
The row index in the global matrix C indicating the first row of sub(C).

jc
(global)
The column index in the global matrix C indicating the first column of
sub(C).

descc
(global and local) array of size dlen_.
The array descriptor for the distributed matrix C.

work
(local)
Workspace array of size lwork.

lwork
(local or global)
The size of the array work.
lwork is local input and must be at least
if side = 'L', lwork ≥ mpc0 + max(1, nqc0),
if side = 'R', lwork ≥ nqc0 + max(max(1, mpc0), numroc(numroc(n + icoffc, nb_a, 0, 0, npcol), nb_a, 0, 0, lcmq)),

where
lcmq = lcm/npcol,
lcm = iclm(nprow, npcol),
iero = mod(ic-1, mb_c),
icoffc = mod(jc-1, nb_c),
icrow = indxg2p(ic, mb_c, myrow, rsrc_c, nprow),
iccol = indxg2p(jc, nb_c, mycol, csr_c, npcol),
Mqc0 = numroc(m+icoffc, nb_c, mycol, icrow, nprow),
Npc0 = numroc(n+icoffc, mb_c, myrow, iccol, npcol),
ilcm, indx2p and numroc are ScaLAPACK tool functions; myrow, mycol, nprow, and npcol can be determined by calling the function blacs_gridinfo.

If lwork = -1, then lwork is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by pxerbla.

**Output Parameters**

**c**

On exit, c is overwritten by Q*sub(C), or Q^T*sub(C)/ Q'*sub(C), or sub(C)*Q, or sub(C)*Q^T / sub(C)*Q'

**work**

On exit, work[0] returns the minimal and optimal lwork.

**info**

(local)

= 0: successful exit
< 0: if the i-th argument is an array and the j-th entry, indexed j-1, had an illegal value,
then info = -(i*100 +j),
if the i-th argument is a scalar and had an illegal value,
then info = -i.

**NOTE**
The distributed submatrices A(ia:*, ja:*) and C(ic:ic+m-1, jc:jc+n-1) must verify some alignment properties, namely the following expressions should be true:

If side = 'L', (mb_a == mb_c) && (iero == icoffc) && (iarow == icrow).

If side = 'R', (mb_a == nb_c) && (iero == icoffc).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?orml2/p?unml2**

Multiplies a general matrix by the orthogonal/unitary matrix from an LQ factorization determined by p?gelqf (unblocked algorithm).

**Syntax**

```c
void psorml2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);

void pdorml2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);

void pcunml2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzunml2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The p?orml2/p?unml2 function overwrites the general real/complex \( m \times n \) distributed matrix sub

\[ C = C(ia:ic+m-1, ja:jc+n-1) \]

with

\[ Q^*C \text{ if } side = 'L' \text{ and } trans = 'N', \text{ or} \]

\[ Q^*C / Q^H C \text{ if } side = 'L' \text{ and } trans = 'T' \text{ (for real flavors) or } trans = 'C' \text{ (for complex flavors), or} \]

\[ \text{sub}(C)Q \text{ if } side = 'R' \text{ and } trans = 'N', \text{ or} \]

\[ \text{sub}(C)Q^T / \text{sub}(C)Q^H \text{ if } side = 'R' \text{ and } trans = 'T' \text{ (for real flavors) or } trans = 'C' \text{ (for complex flavors).} \]

Where \( Q \) is a real orthogonal or complex unitary distributed matrix defined as the product of \( k \) elementary reflectors

\[ Q = H(k)^*...*H(2)^*H(1) \] (for real flavors)

\[ Q = (H(k))^H...*(H(2))^H*(H(1))^H \] (for complex flavors)

As returned by p?gelqf. \( Q \) is of order \( m \) if \( side = 'L' \) and of order \( n \) if \( side = 'R'. \)

**Input Parameters**

- **side** (global)
  - \( 'L' \): apply \( Q \) or \( Q^T \) for real flavors (\( Q^H \) for complex flavors) from the left,
  - \( 'R' \): apply \( Q \) or \( Q^T \) for real flavors (\( Q^H \) for complex flavors) from the right.
**trans**
- 'N': apply $Q$ (no transpose)
- 'T': apply $Q^T$ (transpose, for real flavors)
- 'C': apply $Q^H$ (conjugate transpose, for complex flavors)

**m**
(global)
The number of rows in the distributed matrix sub($C$). $m \geq 0$.

**n**
(global)
The number of columns in the distributed matrix sub($C$). $n \geq 0$.

**k**
(global)
The number of elementary reflectors whose product defines the matrix $Q$.

If $side = 'L'$, $m \geq k \geq 0$;
if $side = 'R'$, $n \geq k \geq 0$.

**a**
(local)
Pointer into the local memory to an array of size 

\[ \text{ldd}_a \times \text{LOCc}(ja+m-1) \] if $side = 'L'$,

\[ \text{ldd}_a \times \text{LOCc}(ja+n-1) \] if $side = 'R'$,

where $\text{ldd}_a \geq \max(1, \text{LOCr}(ia+k-1))$.

On entry, the $i$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by p?gelqf in the $k$ rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$. The argument $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.

**ia**
(global)
The row index in the global matrix $A$ indicating the first row of sub($A$).

**ja**
(global)
The column index in the global matrix $A$ indicating the first column of sub($A$).

**desca**
(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

**tau**
(local)
Array of size $\text{LOCc}(ia+k-1)$. $\tau[i]$ contains the scalar factor of the elementary reflector $H(i+1)$, $i = 0, 1, ..., \text{LOCc}(ja+k-1)-1$, as returned by p?gelqf. This array is tied to the distributed matrix $A$.

**c**
(local)
Pointer into the local memory to an array of size $\text{ldd}_c \times \text{LOCc}(jc+n-1)$. On entry, the local pieces of the distributed matrix sub ($C$).

**ic**
(global)
The row index in the global matrix $C$ indicating the first row of sub($C$).
The column index in the global matrix $C$ indicating the first column of $\text{sub}(C)$.

descC

(global and local) array of size $dlen_\cdot$. The array descriptor for the distributed matrix $C$.

work

(local)

Workspace array of size $lwork$.

lwork

(local or global)

The size of the array $work$.

$lwork$ is local input and must be at least

if $side = 'L'$, $lwork \geq mqc_0 + \max(\max(1, n{pc}_0), \text{numroc}(\text{numroc}(m + \text{icoffc}, mb_\cdot, 0, 0, np{row}), mb_\cdot, 0, 0, lcmp))$,

if $side = 'R'$, $lwork \geq n{pc}_0 + \max(1, mqc_0)$,

where

$lcmp = lcm / np{row},$

$lcm = ilcm(np{row}, np{col}),$

$\text{iroffc} = \text{mod}(ic-1, mb_\cdot),$

$\text{icoffc} = \text{mod}(jc-1, nb_\cdot),$

$\text{icrow} = \text{indxg2p}(ic, mb_\cdot, my{row}, rsr{c}_\cdot, np{row}),$

$\text{iccol} = \text{indxg2p}(jc, nb_\cdot, my{col}, csr{c}_\cdot, np{col}),$

$Mpc_0 = \text{numroc}(m+\text{icoffc}, mb_\cdot, my{col}, \text{icrow}, np{row}),$

$Nqc_0 = \text{numroc}(n+\text{iroffc}, nb_\cdot, my{row}, \text{iccol}, np{col}),$

$ilcm, \text{indxg2p}$ and $\text{numroc}$ are ScaLAPACK tool functions; $my{row}$, $my{col}$, $np{row}$, and $np{col}$ can be determined by calling the function $\text{blacs_gridinfo}$.

If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by $\text{pxerbla}$.

Output Parameters

c

On exit, $c$ is overwritten by $Q\cdot \text{sub}(C)$, or $Q^T\cdot \text{sub}(C)\cdot Q^H\cdot \text{sub}(C)$, or $\text{sub}(C)\cdot Q$, or $\text{sub}(C)\cdot Q^T / \text{sub}(C)\cdot Q^H$.

work

On exit, $work[0]$ returns the minimal and optimal $lwork$.

info

(local)

$= 0$: successful exit

$< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value,

then $info = - (i*100 + j)$,
if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

**NOTE**
The distributed submatrices \( A(ia:*, ja:* \) and \( C(ic:ic+m-1, jc:jc+n-1) \) must verify some alignment properties, namely the following expressions should be true:

If \( side = 'L' \), \((nb_a == mb_c && icoffa == iroffc)\)
If \( side = 'R' \), \((nb_a == nb_c && icoffa == icoffc && icol == icol)\).

**See Also**
Overview for details of ScaLAPACK array descriptor structures and related notations.

\texttt{pormr2/punmr2}

Multiplies a general matrix by the orthogonal/unitary matrix from an RQ factorization determined by \texttt{pgerqf} (unblocked algorithm).

**Syntax**

```c
void psormr2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *tau, float *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, float *work, MKL_INT *lwork, MKL_INT *info);
void pdormr2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *tau, double *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, double *work, MKL_INT *lwork, MKL_INT *info);
void pcunmr2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *tau, MKL_Complex8 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzunmr2 (char *side, char *trans, MKL_INT *m, MKL_INT *n, MKL_INT *k, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *tau, MKL_Complex16 *c, MKL_INT *ic, MKL_INT *jc, MKL_INT *descc, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**
- \texttt{mkl_scalapack.h}

**Description**
The \texttt{pormr2/punmr2} function overwrites the general real/complex \( m \)-by-\( n \) distributed matrix sub
(C)\(=C(ic:ic+m-1, jc:jc+n-1) \) with

\( Q^*\text{sub}(C) \) if \( side = 'L' \) and \( trans = 'N' \), or
\( Q^T\text{sub}(C) / Q^H\text{sub}(C) \) if \( side = 'L' \) and \( trans = 'T' \) (for real flavors) or \( trans = 'C' \) (for complex flavors), or

\( \text{sub}(C)Q \) if \( side = 'R' \) and \( trans = 'N' \), or

\( \text{sub}(C)Q^T / \text{sub}(C)Q^H \) if \( side = 'R' \) and \( trans = 'T' \) (for real flavors) or \( trans = 'C' \) (for complex flavors).
where $Q$ is a real orthogonal or complex unitary distributed matrix defined as the product of $k$ elementary reflectors

$Q = H(1)^*H(2)^*...*H(k)$ (for real flavors)

$Q = (H(1))^H(H(2))^H...*(H(k))^H$ (for complex flavors)

as returned by \texttt{pgerqf}. $Q$ is of order $m$ if $\text{side} = 'L'$ and of order $n$ if $\text{side} = 'R'$.

**Input Parameters**

- \textit{side} (global)
  - $'L'$: apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the left,
  - $'R'$: apply $Q$ or $Q^T$ for real flavors ($Q^H$ for complex flavors) from the right.

- \textit{trans} (global)
  - $'N'$: apply $Q$ (no transpose)
  - $'T'$: apply $Q^T$ (transpose, for real flavors)
  - $'C'$: apply $Q^H$ (conjugate transpose, for complex flavors)

- \textit{m} (global)
  The number of rows in the distributed matrix \texttt{sub(C)}. $m \geq 0$.

- \textit{n} (global)
  The number of columns in the distributed matrix \texttt{sub(C)}. $n \geq 0$.

- \textit{k} (global)
  The number of elementary reflectors whose product defines the matrix $Q$.
  If $\text{side} = 'L'$, $m \geq k \geq 0$;
  if $\text{side} = 'R'$, $n \geq k \geq 0$.

- \textit{a} (local)
  Pointer into the local memory to an array of size
  
  $\text{lld}_a \cdot \text{LOCc}(ja+m-1)$ if $\text{side} = 'L'$,
  
  $\text{lld}_a \cdot \text{LOCc}(ja+n-1)$ if $\text{side} = 'R'$,

  where $\text{lld}_a \geq \max (1, \text{LOCr}(ia+k-1))$.

  On entry, the $i$-th row of the matrix stored in $a$ must contain the vector that defines the elementary reflector $H(i)$, $ia \leq i \leq ia+k-1$, as returned by \texttt{pgerqf} in the $k$ rows of its distributed matrix argument $A(ia:ia+k-1, ja:*)$.

  The argument $A(ia:ia+k-1, ja:*)$ is modified by the function but restored on exit.

- \textit{ia} (global)
  The row index in the global matrix $A$ indicating the first row of \texttt{sub(A)}.

- \textit{ja} (global)
The column index in the global matrix $A$ indicating the first column of sub($A$).

$\text{desca}$

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

$\tau$

(local)

Array of size $\text{LOCc}(ia+k-1)$. $\tau[j]$ contains the scalar factor of the elementary reflector $H(j+1)$, $j = 0, 1, \ldots$, $\text{LOCc}(ja+k-1)-1$, as returned by $\text{pgerqf}$. This array is tied to the distributed matrix $A$.

$c$

(local)

Pointer into the local memory to an array of size $lld_c \times \text{LOCc}(jc+n-1)$. On entry, the local pieces of the distributed matrix sub ($C$).

$ic$

(global)

The row index in the global matrix $C$ indicating the first row of sub($C$).

$jc$

(global)

The column index in the global matrix $C$ indicating the first column of sub($C$).

$\text{descc}$

(global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $C$.

$\text{work}$

(local)

Workspace array of size $lwork$.

$lwork$

(local or global)

The size of the array $\text{work}$.

$lwork$ is local input and must be at least

if $side = 'L'$, $lwork \geq \text{mpc0} + \max(\max(1, nqc0), \text{numroc}(\text{numroc}(m + iroffc, mb_a, 0, 0, nprow), mb_a, 0, 0, lcmp))$,

if $side = 'R'$, $lwork \geq nqc0 + \max(1, \text{mpc0})$,

where $lcmp = \text{lcm}/\text{nprow}$,

$lcm = i\text{icl}(\text{nprow}, \text{ncol})$,

$\text{iroffc} = \text{mod}(ic-1, \text{mb}_c)$,

$\text{icoffc} = \text{mod}(jc-1, \text{nb}_c)$,

$\text{icrow} = \text{indxg2p}(ic, \text{mb}_c, \text{myrow}, \text{rsrc}_c, \text{nprow})$,

$\text{iccol} = \text{indxg2p}(jc, \text{nb}_c, \text{mycol}, \text{csrc}_c, \text{ncol})$,

$\text{Mpc0} = \text{numroc}(m + \text{iroffc}, \text{mb}_a, \text{myrow}, \text{icrow}, \text{nprow})$,

$\text{Nqc0} = \text{numroc}(n + \text{icoffc}, \text{nb}_c, \text{mycol}, \text{iccol}, \text{ncol})$,

$l\text{lcm}$, $\text{indxg2p}$ and $\text{numroc}$ are ScaLAPACK tool functions; $\text{myrow}$, $\text{mycol}$, $\text{nprow}$, and $\text{ncol}$ can be determined by calling the function $\text{blacs_gridinfo}$.
If $lwork = -1$, then $lwork$ is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by `pxerbla`.

### Output Parameters

- **$c$**
  
  On exit, $c$ is overwritten by $Q^*_{\text{sub}(C)}$, or $Q^T_{\text{sub}(C)}/Q^*_{\text{sub}(C)}$, or $\text{sub}(C)*Q$, or $\text{sub}(C)*Q^T/\text{sub}(C)*Q^H$

- **$work$**

- **$info$**
  
  (local)
  
  - $= 0$: successful exit
  - $< 0$: if the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value,
    then $info = - (i*100 + j)$,
  - if the $i$-th argument is a scalar and had an illegal value,
    then $info = -i$.

### NOTE

The distributed submatrices $A(ia:*, ja:*)$ and $C(ic:ic+m-1,jc:jc+n-1)$ must verify some alignment properties, namely the following expressions should be true:

- If $side = 'L'$, $(nb_a == mb_c) \&\& (icoffa == iroffc)$.
- If $side = 'R'$, $(nb_a == nb_c) \&\& (icoffa == icoffc) \&\& (iacol == iccol)$.

### See Also

- **Overview** for details of ScaLAPACK array descriptor structures and related notations.

### `p?pbtrsv`

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a banded matrix computed by `p?pbtf`.  

### Syntax

```c
void pspbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
    float *a , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *descb ,
    float *af , MKL_INT *laf , float *work , MKL_INT *lwork , MKL_INT *info );
void pdpbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
    double *a , MKL_INT *ja , MKL_INT *desca , double *b , MKL_INT *ib , MKL_INT *descb ,
    double *af , MKL_INT *laf , double *work , MKL_INT *lwork , MKL_INT *info );
void pcpbtrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *bw , MKL_INT *nrhs ,
    MKL_Complex8 *a , MKL_INT *ja , MKL_INT *desca , MKL_Complex8 *b , MKL_INT *ib ,
    MKL_INT *descb , MKL_Complex8 *af , MKL_INT *laf , MKL_Complex8 *work , MKL_INT *
    lwork , MKL_INT *info );
```
void pzpbtrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *bw, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);

Include Files
• mkl_scalapack.h

Description
The \texttt{p?pbtrsv} function solves a banded triangular system of linear equations
\[ A(1:n, ja:ja+n-1)X = B(jb:jb+n-1, 1:nrhs) \]
or
\[ A(1:n, ja:ja+n-1)^T X = B(jb:jb+n-1, 1:nrhs) \]
for real flavors,
\[ A(1:n, ja:ja+n-1)^H X = B(jb:jb+n-1, 1:nrhs) \]
for complex flavors,
where \( A(1:n, ja:ja+n-1) \) is a banded triangular matrix factor produced by the Cholesky factorization code \texttt{p?pbtrf} and is stored in \( A(1:n, ja:ja+n-1) \) and \( af \). The matrix stored in \( A(1:n, ja:ja+n-1) \) is either upper or lower triangular according to \( uplo \).

The function \texttt{p?pbtrf} must be called first.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\begin{itemize}
\item \textbf{uplo} (global) Must be 'U' or 'L'.
\begin{itemize}
\item If \texttt{uplo} = 'U', upper triangle of \( A(1:n, ja:ja+n-1) \) is stored;
\item If \texttt{uplo} = 'L', lower triangle of \( A(1:n, ja:ja+n-1) \) is stored.
\end{itemize}
\end{itemize}

\item \textbf{trans} (global) Must be 'N' or 'T' or 'C'.
\begin{itemize}
\item If \texttt{trans} = 'N', solve with \( A(1:n, ja:ja+n-1) \);
\item If \texttt{trans} = 'T' or 'C' for real flavors, solve with \( A(1:n, ja:ja+n-1)^T \).
\item If \texttt{trans} = 'C' for complex flavors, solve with conjugate transpose \( (A(1:n, ja:ja+n-1))^H \).
\end{itemize}

\item \textbf{n} (global) The number of rows and columns to be operated on, that is, the order of the distributed submatrix \( A(1:n, ja:ja+n-1) \). \( n \geq 0 \).

\item \textbf{bw} (global)
The number of subdiagonals in 'L' or 'U', $0 \leq bw \leq n-1$.

$nrhs$ (global)
The number of right hand sides; the number of columns of the distributed submatrix $B(jb:jb+n-1, 1:nrhs); nrhs \geq 0$.

$a$ (local)
Pointer into the local memory to an array with the first size $lld_a \geq (bw +1)$, stored in desca.

On entry, this array contains the local pieces of the $n$-by-$n$ symmetric banded distributed Cholesky factor $L$ or $L^*A(1:n, ja:ja+n-1)$.

This local portion is stored in the packed banded format used in LAPACK. See the Application Notes below and the ScaLAPACK manual for more detail on the format of distributed matrices.

$ja$ (global) The index in the global in the global matrix $A$ that points to the start of the matrix to be operated on (which may be either all of $A$ or a submatrix of $A$).

desca (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $A$.

If 1D type ($dtype_a = 501$), then $dlen \geq 7$;

If 2D type ($dtype_a = 1$), then $dlen \geq 9$.

Contains information on mapping of $A$ to memory. (See ScaLAPACK manual for full description and options.)

$b$ (local)
Pointer into the local memory to an array of local lead size $lld_b \geq nb$.

On entry, this array contains the local pieces of the right hand sides $B(jb:jb+n-1, 1:nrhs)$.

$ib$ (global) The row index in the global matrix $B$ that points to the first row of the matrix to be operated on (which may be either all of $B$ or a submatrix of $B$).

descb (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix $B$.

If 1D type ($dtype_b = 502$), then $dlen \geq 7$;

If 2D type ($dtype_b = 1$), then $dlen \geq 9$.

Contains information on mapping of $B$ to memory. Please, see ScaLAPACK manual for full description and options.

$laf$ (local)
The size of user-input auxiliary fill-in space $af$. Must be $laf \geq (nb +2* bw) * bw$. If $laf$ is not large enough, an error code will be returned and the minimum acceptable size will be returned in $af[0]$.

$work$ (local)
The array \textit{work} is a temporary workspace array of size \textit{lwork}. This space may be overwritten in between function calls.

\textit{lwork} (local or global) The size of the user-input workspace \textit{work}, must be at least \textit{lwork} \geq \textit{bw} \times \textit{nrhs}. If \textit{lwork} is too small, the minimal acceptable size will be returned in \textit{work[0]} and an error code is returned.

**Output Parameters**

\textit{af} (local)

The array \textit{af} is of size \textit{laf}. It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function \texttt{p?pbtrf} and is stored in \textit{af}. If a linear system is to be solved using \texttt{p?pbtrs} after the factorization function, \textit{af} must not be altered after the factorization.

\textit{b}

On exit, this array contains the local piece of the solutions distributed matrix \(X\).

\textit{work[0]}

On exit, \textit{work[0]} contains the minimum value of \textit{lwork}.

\textit{info} (local)

\[ \begin{align*}
\text{info} = 0 : & \text{ successful exit} \\
\text{< 0} : & \text{ if the } i\text{-th argument is an array and the } j\text{-th entry, indexed } j-1, \text{ had an illegal value,} \\
& \text{ then } info = -(i \times 100 + j), \\
\text{ if the } i\text{-th argument is a scalar and had an illegal value,} \\
& \text{ then } info = -i.
\end{align*} \]

**Application Notes**

If the factorization function and the solve function are to be called separately to solve various sets of right-hand sides using the same coefficient matrix, the auxiliary space \textit{af} must not be altered between calls to the factorization function and the solve function.

The best algorithm for solving banded and tridiagonal linear systems depends on a variety of parameters, especially the bandwidth. Currently, only algorithms designed for the case \(N/P \gg \text{bw}\) are implemented. These algorithms go by many names, including Divide and Conquer, Partitioning, domain decomposition-type, etc.

The Divide and Conquer algorithm assumes the matrix is narrowly banded compared with the number of equations. In this situation, it is best to distribute the input matrix \(A\) one-dimensionally, with columns atomic and rows divided amongst the processes. The basic algorithm divides the banded matrix up into \(P\) pieces with one stored on each processor, and then proceeds in 2 phases for the factorization or 3 for the solution of a linear system.

1. **Local Phase**: The individual pieces are factored independently and in parallel. These factors are applied to the matrix creating fill-in, which is stored in a non-inspectable way in auxiliary space \textit{af}. Mathematically, this is equivalent to reordering the matrix \(A\) as \(P\text{AT}\) and then factoring the principal leading submatrix of size equal to the sum of the sizes of the matrices factored on each processor. The factors of these submatrices overwrite the corresponding parts of \(A\) in memory.

2. **Reduced System Phase**: A small \((\text{bw} \times (P-1))\) system is formed representing interaction of the larger blocks and is stored (as are its factors) in the space \textit{af}. A parallel Block Cyclic Reduction algorithm is used. For a linear system, a parallel front solve followed by an analogous backsolve, both using the structure of the factored matrix, are performed.
3. **Back Substitution Phase**: For a linear system, a local backsubstitution is performed on each processor in parallel.

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**p?pttrsv**

Solves a single triangular linear system via frontsolve or backsolve where the triangular matrix is a factor of a tridiagonal matrix computed by **p?pttrf**.

**Syntax**

```c
void pspttrsv (char *uplo, MKL_INT *n, MKL_INT *nrhs, float *d, float *e, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *descb, float *af, MKL_INT *laf, float *work, MKL_INT *lwork, MKL_INT *info);

void pdpttrsv (char *uplo, MKL_INT *n, MKL_INT *nrhs, double *d, double *e, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *descb, double *af, MKL_INT *laf, double *work, MKL_INT *lwork, MKL_INT *info);

void pcpttrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, float *d, MKL_Complex8 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex8 *af, MKL_INT *laf, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);

void pzpttrsv (char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, double *d, MKL_Complex16 *e, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *descb, MKL_Complex16 *af, MKL_INT *laf, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The **p?pttrsv** function solves a tridiagonal triangular system of linear equations

\[ A(1:n, ja:ja+n-1)X = B(jb:jb+n-1, 1:nrhs) \]

or

\[ A(1:n, ja:ja+n-1)^T X = B(jb:jb+n-1, 1:nrhs) \] for real flavors,

\[ A(1:n, ja:ja+n-1)^H X = B(jb:jb+n-1, 1:nrhs) \] for complex flavors,

where \( A(1:n, ja:ja+n-1) \) is a tridiagonal triangular matrix factor produced by the Cholesky factorization code **p?pttrf** and is stored in \( A(1:n, ja:ja+n-1) \) and \( af \). The matrix stored in \( A(1:n, ja:ja+n-1) \) is either upper or lower triangular according to \( uplo \).

The function **p?pttrf** must be called first.

**Input Parameters**

- **uplo** (global) Must be 'U' or 'L'.
  - If **uplo** = 'U', upper triangle of \( A(1:n, ja:ja+n-1) \) is stored;
  - If **uplo** = 'L', lower triangle of \( A(1:n, ja:ja+n-1) \) is stored.
trans

(global) Must be 'N' or 'C'.

If \( \text{trans} = 'N' \), solve with \( A(1:n, ja:ja+n-1) \); 

If \( \text{trans} = 'C' \) (for complex flavors), solve with conjugate transpose 
\( (A(1:n, ja:ja+n-1))^H \).

\( n \)

(global)

The number of rows and columns to be operated on, that is, the order of 
the distributed submatrix \( A(1:n, ja:ja+n-1) \). \( n \geq 0 \).

\( nrhs \)

(global)

The number of right hand sides; the number of columns of the distributed 
submatrix \( B(jb:jb+n-1, 1:nrhs) \); \( nrhs \geq 0 \).

\( d \)

(local)

Pointer to the local part of the global vector storing the main diagonal of the 
matrix; must be of size \( \geq nb_a \).

\( e \)

(local)

Pointer to the local part of the global vector \( du \) storing the upper diagonal of 
the matrix; must be of size \( \geq nb_a \). Globally, \( du(n) \) is not referenced, and \( du \) 
must be aligned with \( d \).

\( ja \)

(global) The index in the global matrix \( A \) that points to the start of the 
matrix to be operated on (which may be either all of \( A \) or a submatrix of \( A \)).

\( desca \)

(global and local) array of size \( dlen_\). The array descriptor for the 
distributed matrix \( A \).

If 1D type (\( dtype_a = 501 \) or 502), then \( dlen \geq 7 \); 

If 2D type (\( dtype_a = 1 \)), then \( dlen \geq 9 \).

Contains information on mapping of \( A \) to memory. See ScaLAPACK manual 
for full description and options.

\( b \)

(local)

Pointer into the local memory to an array of local lead size \( lld_b \geq nb \).

On entry, this array contains the local pieces of the right hand sides 
\( B(jb:jb+n-1, 1:nrhs) \).

\( ib \)

(global) The row index in the global matrix \( B \) that points to the first row of 
the matrix to be operated on (which may be either all of \( B \) or a submatrix of 
\( B \)).

\( descb \)

(global and local) array of size \( dlen_\). The array descriptor for the 
distributed matrix \( B \).

If 1D type (\( dtype_b = 502 \)), then \( dlen \geq 7 \); 

If 2D type (\( dtype_b = 1 \)), then \( dlen \geq 9 \).

Contains information on mapping of \( B \) to memory. See ScaLAPACK manual 
for full description and options.

\( laf \)

(local)
The size of user-input auxiliary fill-in space \( af \). Must be \( laf \geq (nb + 2 * bw) * bw \).

If \( laf \) is not large enough, an error code will be returned and the minimum acceptable size will be returned in \( af[0] \).

\( \text{work} \)

(local)

The array \( \text{work} \) is a temporary workspace array of size \( lwork \). This space may be overwritten in between function calls.

\( lwork \)

(local or global) The size of the user-input workspace \( \text{work} \), must be at least \( lwork \geq (10 + 2 \times \min(100, nrhs)) \times npcol + 4 \times nrhs \). If \( lwork \) is too small, the minimal acceptable size will be returned in \( \text{work}[0] \) and an error code is returned.

Output Parameters

\( d, e \)

(local).

On exit, these arrays contain information on the factors of the matrix.

\( af \)

(local)

The array \( af \) is of size \( laf \). It contains auxiliary fill-in space. The fill-in space is created in a call to the factorization function \( p?pbtrf \) and is stored in \( af \). If a linear system is to be solved using \( p?pttrs \) after the factorization function, \( af \) must not be altered after the factorization.

\( b \)

On exit, this array contains the local piece of the solutions distributed matrix \( X \).

\( \text{work}[0] \)

On exit, \( \text{work}[0] \) contains the minimum value of \( lwork \).

\( \text{info} \)

(local)

\( = 0 \): successful exit
\( < 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value,
then \( \text{info} = -(i*100 + j) \),
if the \( i \)-th argument is a scalar and had an illegal value,
then \( \text{info} = -i \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\( p?potf2 \)

Computes the Cholesky factorization of a symmetric/Hermitian positive definite matrix (local unblocked algorithm).

Syntax

```c
void pspotf2 (char *uplo , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , MKL_INT *info );
```
void pdpotf2 (char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pcpotf2 (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pzpotf2 (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);

Include Files
- mkl_scalapack.h

Description
The p?potf2 function computes the Cholesky factorization of a real symmetric or complex Hermitian positive definite distributed matrix sub(A) = A(ia:ia+n-1, ja:ja+n-1).

The factorization has the form
sub(A) = U'*U, if uplo = 'U', or sub(A) = L*L', if uplo = 'L',
where U is an upper triangular matrix, L is lower triangular. X' denotes transpose (conjugate transpose) of X.

Input Parameters

uplo
   (global)
   Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix A is stored.
   = 'U': upper triangle of sub(A) is stored;
   = 'L': lower triangle of sub(A) is stored.

n
   (global)
   The number of rows and columns to be operated on, that is, the order of the distributed matrix sub(A). n ≥ 0.

a
   (local)
   Pointer into the local memory to an array of size lld_a * LOCc(ja+n-1) containing the local pieces of the n-by-n symmetric distributed matrix sub(A) to be factored.
   If uplo = 'U', the leading n-by-n upper triangular part of sub(A) contains the upper triangular matrix and the strictly lower triangular part of this matrix is not referenced.
   If uplo = 'L', the leading n-by-n lower triangular part of sub(A) contains the lower triangular matrix and the strictly upper triangular part of sub(A) is not referenced.

ia, ja
   (global)
   The row and column indices in the global matrix A indicating the first row and the first column of the sub(A), respectively.

desca
   (global and local) array of size dlen_. The array descriptor for the distributed matrix A.
Output Parameters

a
(local)
On exit,
if uplo = 'U', the upper triangular part of the distributed matrix contains
the Cholesky factor U;
if uplo = 'L', the lower triangular part of the distributed matrix contains
the Cholesky factor L.

info
(local)
= 0: successful exit
< 0: if the i-th argument is an array and the j-th entry, indexed j-1, had an
illegal value,
then info = - (i*100 +j),
if the i-th argument is a scalar and had an illegal value,
then info = -i.
> 0: if info = k, the leading minor of order k is not positive definite, and
the factorization could not be completed.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?rot
Applies a planar rotation to two distributed vectors.

Syntax
void psrot(MKL_INT* n, float* x, MKL_INT* ix, MKL_INT* jx, MKL_INT* descx, MKL_INT*
inx, float* y, MKL_INT* iy, MKL_INT* jy, MKL_INT* descy, MKL_INT* incy, float* cs,
float* sn, float* work, MKL_INT* lwork, MKL_INT* info);
void pdrot(MKL_INT* n, double* x, MKL_INT* ix, MKL_INT* jx, MKL_INT* descx, MKL_INT*
inx, double* y, MKL_INT* iy, MKL_INT* jy, MKL_INT* descy, MKL_INT* incy, double* cs,
double* sn, double* work, MKL_INT* lwork, MKL_INT* info);

Include Files
• mkl_scalapack.h

Description
p?rot applies a planar rotation defined by cs and sn to the two distributed vectors sub(x) and sub(y).

Input Parameters

n (global )
The number of elements to operate on when applying the planar rotation to
x and y (n≥0).

x (local) array of size ( (jx-1)*m_x + ix + ( n - 1 )*abs( incx ))
This array contains the entries of the distributed vector sub( x ).

ix (global )
The global row index of the submatrix of the distributed matrix x to operate on. If incx = 1, then it is required that ix = iy. 1 ≤ ix ≤ m_x.

jx (global )
The global column index of the submatrix of the distributed matrix x to operate on. If incx = m_x, then it is required that jx = jy. 1 ≤ jx ≤ n_x.

descx (global and local) array of size 9
The array descriptor of the distributed matrix x.

incx (global )
The global increment for the elements of x. Only two values of incx are supported in this version, namely 1 and m_x. Moreover, it must hold that incx = m_x if incy = m_y and that incx = 1 if incy = 1.

y (local) array of size ( (jy-1)*m_y + iy + (n - 1)*abs( incy ) )
This array contains the entries of the distributed vector sub( y ).

iy (global )
The global row index of the submatrix of the distributed matrix y to operate on. If incy = 1, then it is required that iy = ix. 1 ≤ iy ≤ m_y.

jy (global )
The global column index of the submatrix of the distributed matrix y to operate on. If incy = m_x, then it is required that jy = jx. 1 ≤ jy ≤ m_y.

descy (global and local) array of size 9
The array descriptor of the distributed matrix y.

incy (global )
The global increment for the elements of y. Only two values of incy are supported in this version, namely 1 and m_y. Moreover, it must hold that incy = m_y if incx = m_x and that incy = 1 if incx = 1.

cs, sn (global)
The parameters defining the properties of the planar rotation. It must hold that 0 ≤ cs, sn ≤ 1 and that sn^2 + cs^2 = 1. The latter is hardly checked in finite precision arithmetics.

work (local workspace) array of size lwork

lwork (local )
The length of the workspace array work.
If incx = 1 and incy = 1, then lwork = 2*m_x
If \( lwork = -1 \), then a workspace query is assumed; the function only calculates the optimal size of the \( work \) array, returns this value as the first entry of the IWORK array, and no error message related to LIWORK is issued by \texttt{pxerbla}.

**OUTPUT Parameters**

\( x \)
\( y \)
\( work[0] \)
\( info \)

On exit, if \( info = 0 \), \( work[0] \) returns the optimal \( lwork \) (global).

\( = 0 \): successful exit
\( < 0 \): if \( info = -i \), the \( i \)-th argument had an illegal value.

If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( info = -(i*100+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?rscl**

*Multiplies a vector by the reciprocal of a real scalar.*

**Syntax**

```c
void psrscl (MKL_INT *n, float *sa, float *sx, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
void pdrscl (MKL_INT *n, double *sa, double *sx, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
void pcsrscl (MKL_INT *n, float *sa, MKL_Complex8 *sx, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
void pzdrscl (MKL_INT *n, double *sa, MKL_Complex16 *sx, MKL_INT *ix, MKL_INT *jx, MKL_INT *descx, MKL_INT *incx);
```

**Include Files**

- \texttt{mkl_scalapack.h}

**Description**

The \( p?rscl \) function multiplies an \( n \)-element real/complex vector \( \text{sub}(X) \) by the real scalar \( 1/a \). This is done without overflow or underflow as long as the final result \( \text{sub}(X)/a \) does not overflow or underflow.

\( \text{sub}(X) \) denotes \( X(ix:ix+n-1, jx:jx) \), if \( incx = 1 \),
and \( X(ix:ix, jx:jx+n-1) \), if \( incx = m_x \).

**Input Parameters**

\( n \) (global)
The number of components of the distributed vector \( sub(X) \). \( n \geq 0 \).

\( sa \)
The scalar \( a \) that is used to divide each component of the vector \( sub(X) \). This parameter must be \( \geq 0 \).

\( sx \)
Array containing the local pieces of a distributed matrix of size of at least 
\(((jx-1)*m_x + ix + (n-1)*abs(incx)). This array contains the entries 
of the distributed vector \( sub(X) \).

\( ix \)
(global) The row index of the submatrix of the distributed matrix \( X \) to 
operate on.

\( jx \)
(global)
The column index of the submatrix of the distributed matrix \( X \) to operate 
on.

\( descx \)
(global and local)
Array of size 9. The array descriptor for the distributed matrix \( X \).

\( incx \)
(global)
The increment for the elements of \( X \). This version supports only two values 
of \( incx \), namely 1 and \( m_x \).

**Output Parameters**

\( sx \)
On exit, the result \( x/a \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?sygs2/p?hegs2**

Reduces a symmetric/Hermitian positive-definite 
generalized eigenproblem to standard form, using the 
factorization results obtained from p?potrf (local 
unblocked algorithm).

**Syntax**

```c
void pssygs2 (MKL_INT *ibtype, char *uplo, MKL_INT *n, float *a, MKL_INT *ia, 
MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, 
MKL_INT *info );
void pdsygs2 (MKL_INT *ibtype, char *uplo, MKL_INT *n, double *a, MKL_INT *ia, 
MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, 
MKL_INT *info );
void pchegs2 (MKL_INT *ibtype, char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT 
*ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, 
MKL_INT *descb, MKL_INT *info );
void pzhegs2 (MKL_INT *ibtype, char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT 
*ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, 
MKL_INT *descb, MKL_INT *info );
```
Include Files

- mkl_scalapack.h

Description

The `p?sygs2/ p?hegs2` function reduces a real symmetric-definite or a complex Hermitian positive-definite generalized eigenproblem to standard form.

Here `sub(A)` denotes `A(ia:ia+n-1, ja:ja+n-1)`, and `sub(B)` denotes `B(ib:ib+n-1, jb:jb+n-1)`.

If `ibtype = 1`, the problem is

\[
\text{sub}(A)*x = \lambda*\text{sub}(B)*x
\]

and `sub(A)` is overwritten by

\[
\text{inv}(U^T)*\text{sub}(A)*\text{inv}(U) \quad \text{or} \quad \text{inv}(L)*\text{sub}(A)*\text{inv}(L^T)
\]

- for real flavors, and

\[
\text{inv}(U^H)*\text{sub}(A)*\text{inv}(U) \quad \text{or} \quad \text{inv}(L)*\text{sub}(A)*\text{inv}(L^H)
\]

- for complex flavors.

If `ibtype = 2` or `3`, the problem is

\[
\text{sub}(A)*\text{sub}(B)*x = \lambda*x \quad \text{or} \quad \text{sub}(B)*\text{sub}(A)*x = \lambda*x
\]

and `sub(A)` is overwritten by

\[
U*\text{sub}(A)*U^T \quad \text{or} \quad L*L^T*\text{sub}(A)*L
\]

- for real flavors and

\[
U*\text{sub}(A)*U^H \quad \text{or} \quad L*L^H*\text{sub}(A)*L
\]

- for complex flavors.

The matrix `sub(B)` must have been previously factorized as `U^T*U` or `L*L^T` (for real flavors), or as `U^H*U` or `L*L^H` (for complex flavors) by `p?potrf`.

Input Parameters

- `ibtype` (global)
  
  `= 1:`
  
  compute `inv(U^T)*sub(A)*inv(U)`, or `inv(L)*sub(A)*inv(L^T)` for real functions,
  
  and `inv(U^H)*sub(A)*inv(U)`, or `inv(L)*sub(A)*inv(L^H)` for complex functions;
  
  `= 2` or `3`:
  
  compute `U*sub(A)*U^T`, or `L*L^T*sub(A)*L` for real functions,
  
  and `U*sub(A)*U^H` or `L*L^H*sub(A)*L` for complex functions.

- `uplo` (global)

  Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix `sub(A)` is stored, and how `sub(B)` is factorized.

  `= 'U'`: Upper triangular of `sub(A)` is stored and `sub(B)` is factorized as `U^T*U` (for real functions) or as `U^H*U` (for complex functions).

  `= 'L'`: Lower triangular of `sub(A)` is stored and `sub(B)` is factorized as `L*L^T` (for real functions) or as `L*L^H` (for complex functions)

- `n` (global)

  The order of the matrices `sub(A)` and `sub(B)`. \( n \geq 0 \).
**Input Parameters**

\( a \) (local)

Pointer into the local memory to an array of size \( lld_a \ast LOCc(ja+n-1) \).

On entry, this array contains the local pieces of the \( n \)-by-\( n \) symmetric/Hermitian distributed matrix sub(\( A \)).

If \( uplo = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of sub(\( A \)) contains the upper triangular part of the matrix, and the strictly lower triangular part of sub(\( A \)) is not referenced.

If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of sub(\( A \)) contains the lower triangular part of the matrix, and the strictly upper triangular part of sub(\( A \)) is not referenced.

\( ia, ja \) (global)

The row and column indices in the global matrix \( A \) indicating the first row and the first column of the sub(\( A \)), respectively.

\( desca \) (global and local) array of size \( dlen_\ast \). The array descriptor for the distributed matrix \( A \).

\( B \) (local)

Pointer into the local memory to an array of size \( lld_b \ast LOCc(jb+n-1) \).

On entry, this array contains the local pieces of the triangular factor from the Cholesky factorization of sub(\( B \)) as returned by \( p?potrf \).

\( ib, jb \) (global)

The row and column indices in the global matrix \( B \) indicating the first row and the first column of the sub(\( B \)), respectively.

\( descb \) (global and local) array of size \( dlen_\ast \). The array descriptor for the distributed matrix \( B \).

**Output Parameters**

\( a \) (local)

On exit, if \( info = 0 \), the transformed matrix is stored in the same format as sub(\( A \)).

\( info \)

\( = 0 \): successful exit.

\(< 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value,

then \( info = - (i \ast 100 + j) \),

if the \( i \)-th argument is a scalar and had an illegal value,

then \( info = -i \).

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?sytd2/p?hetd2**

Reduces a symmetric/Hermitian matrix to real symmetric tridiagonal form by an orthogonal/unitary similarity transformation (local unblocked algorithm).

**Syntax**

```c
void pssytd2 (char *uplo, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, float *tau, float *work, MKL_INT *lwork, MKL_INT *info);
void pdsytd2 (char *uplo, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, double *tau, double *work, MKL_INT *lwork, MKL_INT *info);
void pchetd2 (char *uplo, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *d, float *e, MKL_Complex8 *tau, MKL_Complex8 *work, MKL_INT *lwork, MKL_INT *info);
void pzhetd2 (char *uplo, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *d, double *e, MKL_Complex16 *tau, MKL_Complex16 *work, MKL_INT *lwork, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?sytd2/p?hetd2` function reduces a real symmetric/complex Hermitian matrix `sub(A)` to symmetric/Hermitian tridiagonal form `T` by an orthogonal/unitary similarity transformation:

\[ Q^* sub(A) Q = T, \]

where `sub(A) = A(ia:ia+n-1, ja:ja+n-1)`.

**Input Parameters**

- `uplo` (global)
  
  Specifies whether the upper or lower triangular part of the symmetric/Hermitian matrix `sub(A)` is stored:
  
  - `'U'`: upper triangular
  - `'L'`: lower triangular

- `n` (global)
  
  The number of rows and columns to be operated on, that is, the order of the distributed matrix `sub(A)`. \( n \geq 0 \).

- `a` (local)
  
  Pointer into the local memory to an array of size \( lld_a \times LOC_c(ja+n-1) \).
  
  On entry, this array contains the local pieces of the \( n \)-by-\( n \) symmetric/Hermitian distributed matrix `sub(A)`.
  
  If `uplo = 'U'`, the leading \( n \)-by-\( n \) upper triangular part of `sub(A)` contains the upper triangular part of the matrix, and the strictly lower triangular part of `sub(A)` is not referenced.
If \( uplo = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of \( sub(A) \) contains the lower triangular part of the matrix, and the strictly upper triangular part of \( sub(A) \) is not referenced.

\( ia, ja \)  
(global)  
The row and column indices in the global matrix \( A \) indicating the first row and the first column of the \( sub(A) \), respectively.

\( desca \)  
(global and local) array of size \( dlen_\). The array descriptor for the distributed matrix \( A \).

\( work \)  
(local)  
The array \( work \) is a temporary workspace array of size \( lwork \).

**Output Parameters**

\( a \)  
On exit, if \( uplo = 'U' \), the diagonal and first superdiagonal of \( sub(A) \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements above the first superdiagonal, with the array \( tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors; if \( uplo = 'L' \), the diagonal and first subdiagonal of \( A \) are overwritten by the corresponding elements of the tridiagonal matrix \( T \), and the elements below the first subdiagonal, with the array \( tau \), represent the orthogonal/unitary matrix \( Q \) as a product of elementary reflectors. See the Application Notes below.

\( d \)  
(local)  
Array of size \( LOCc(ja+n-1) \). The diagonal elements of the tridiagonal matrix \( T \):

\[
d[i] = A(i+1,i+1), \text{ where } i=0,1,..., LOCc(ja+n-1)-1 ; \ d \text{ is tied to the distributed matrix } A.
\]

\( e \)  
(local)  
Array of size \( LOCc(ja+n-1) \),

\[
e[i] = A(i+1,i+2) \text{ if } uplo = 'U',
\]

\[
e[i] = A(i+2,i+1) \text{ if } uplo = 'L',
\]

where \( i=0,1,..., LOCc(ja+n-1)-1 \). \( e \) is tied to the distributed matrix \( A \).

\( tau \)  
(local)  
Array of size \( LOCc(ja+n-1) \).

The scalar factors of the elementary reflectors. \( tau \) is tied to the distributed matrix \( A \).

\( work[0] \)  
On exit, \( work[0] \) returns the minimal and optimal value of \( lwork \).

\( lwork \)  
(local or global)
The size of the workspace array \textit{work}.

\textit{lwork} is local input and must be at least \textit{lwork} \geq 3n.

If \textit{lwork} = -1, then \textit{lwork} is global input and a workspace query is assumed; the function only calculates the minimum and optimal size for all work arrays. Each of these values is returned in the first entry of the corresponding work array, and no error message is issued by \texttt{pxerbla}.

\textit{info} (local)
= 0: successful exit
< 0: if the \textit{i}-th argument, indexed \textit{i-1}, is an array and the \textit{j}-th entry had an illegal value,
then \textit{info} = -(\textit{i}*100+\textit{j}),
if the \textit{i}-th argument is a scalar and had an illegal value,
then \textit{info} = -\textit{i}.

\textbf{Application Notes}

If \textit{uplo} = 'U', the matrix \(Q\) is represented as a product of elementary reflectors

\(Q = H(n-1)\ldots H(2)H(1)\)

Each \(H(i)\) has the form

\(H(i) = I - \tau v v'\),

where \(\tau\) is a real/complex scalar, and \(v\) is a real/complex vector with \(v(i+1:n) = 0\) and \(v(i) = 1\);

\(v(1:i-1)\) is stored on exit in \(A(ia:ia+i-2, ja+i)\), and \(\tau\) in \(\tau[ja+i-2]\).

If \textit{uplo} = 'L', the matrix \(Q\) is represented as a product of elementary reflectors

\(Q = H(1)H(2)\ldots H(n-1)\).

Each \(H(i)\) has the form

\(H(i) = I - \tau v v'\),

where \(\tau\) is a real/complex scalar, and \(v\) is a real/complex vector with \(v(1:i) = 0\) and \(v(i+1) = 1\); \(v(i+2:n)\) is stored on exit in \(A(ia+i+1:ia+n-1, ja+i-1)\), and \(\tau\) in \(\tau[ja+i-2]\).

The contents of sub \((A)\) on exit are illustrated by the following examples with \(n = 5\):

\[ \begin{bmatrix} d & e & v_2 & v_3 & v_4 \\ d & e & v_3 & v_4 \\ d & e & v_4 \\ d & e \\ e \end{bmatrix} \quad \begin{bmatrix} d \\ e \\ v_1 & e & d \\ v_1 & v_2 & e & d \\ v_1 & v_2 & v_3 & e & d \end{bmatrix} \]

where \(d\) and \(e\) denotes diagonal and off-diagonal elements of \(T\), and \(v_i\) denotes an element of the vector defining \(H(i)\).
NOTE
The distributed matrix sub(A) must verify some alignment properties, namely the following expression should be true:

\[(mb_a==nb_a && iroffa==icoffa)\]

where \(iroffa = \text{mod}(ia - 1, mb_a)\) and \(icoffa = \text{mod}(ja -1, nb_a)\).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

**p?trord**
*Reorders the Schur factorization of a general matrix.*

**Syntax**

```c
void pstrord(char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, float* t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, float* q, MKL_INT* iq, MKL_INT* jq, MKL_INT* descq, float* wr, float* wi, MKL_INT* m, float* work, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
void pdtrord(char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, double* t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, double* q, MKL_INT* iq, MKL_INT* jq, MKL_INT* descq, double* wr, double* wi, MKL_INT* m, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

`p?trord` reorders the real Schur factorization of a real matrix \(A = Q^*TQ^\top\), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix \(T\), and the leading columns of \(Q\) form an orthonormal basis of the corresponding right invariant subspace.

\(T\) must be in Schur form (as returned by `p?lahqr`), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

This function uses a delay and accumulate procedure for performing the off-diagonal updates.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

- `compq` (global)
= 'V': update the matrix q of Schur vectors;
= 'N': do not update q.

select (global) array of size n

select specifies the eigenvalues in the selected cluster. To select a real eigenvalue \( w(j) \), select[\( j-1 \)] must be set to 1. To select a complex conjugate pair of eigenvalues \( w(j) \) and \( w(j+1) \), corresponding to a 2-by-2 diagonal block, either select[\( j-1 \)] or select[\( j \)] or both must be set to 1; a complex conjugate pair of eigenvalues must be either both included in the cluster or both excluded.

para (global)

Block parameters:

\( para[0] \) maximum number of concurrent computational windows allowed in the algorithm; \( 0 < para[0] \leq \min(nprow, ncol) \) must hold;

\( para[1] \) number of eigenvalues in each window; \( 0 < para[1] < para[2] \) must hold;


\( para[3] \) minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations; \( 0 \leq para[3] \leq 100 \) must hold;

\( para[4] \) width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form; \( 0 < para[4] \leq mb_t \) must hold.

\( para[5] \) the maximum number of eigenvalues moved together over a process border; in practice, this will be approximately half of the cross border window size; \( 0 < para[5] \leq para[1] \) must hold.

n (global)
The order of the globally distributed matrix \( t \). \( n \geq 0 \).

t (local) array of size \( lld_t \times LOC_c(n) \).
The local pieces of the global distributed upper quasi-triangular matrix \( T \), in Schur form.

it, jt (global)
The row and column index in the global matrix \( T \) indicating the first column of \( T \). \( it = jt = 1 \) must hold (see Application Notes).

desct (global and local) array of size \( dlen_\_ \).
The array descriptor for the global distributed matrix \( T \).

q (local) array of size \( lld_q \times LOC_c(n) \).
On entry, if \( \text{compq} = 'V' \), the local pieces of the global distributed matrix \( Q \) of Schur vectors.

If \( \text{compq} = 'N' \), \( q \) is not referenced.

\( iq, jq \)  
(global)  
The column index in the global matrix \( Q \) indicating the first column of \( Q \). \( iq = jq = 1 \) must hold (see Application Notes).

\( \text{descq} \)  
(global and local) array of size \( dlen_\).  
The array descriptor for the global distributed matrix \( Q \).

\( \text{work} \)  
(local workspace) array of size \( lwork \)

\( lwork \)  
(local)  
The size of the array \( \text{work} \).

If \( lwork = -1 \), then a workspace query is assumed; the function only calculates the optimal size of the \( \text{work} \) array, returns this value as the first entry of the \( \text{work} \) array, and no error message related to \( lwork \) is issued by \text{pxerbla}.

\( iwork \)  
(local workspace) array of size \( liwork \)

\( liwork \)  
(local)  
The size of the array \( iwork \).

If \( liwork = -1 \), then a workspace query is assumed; the function only calculates the optimal size of the \( iwork \) array, returns this value as the first entry of the \( iwork \) array, and no error message related to \( liwork \) is issued by \text{pxerbla}.

**OUTPUT Parameters**

\( \text{select} \)  
(global) array of size \( n \)  
The (partial) reordering is displayed.

\( t \)  
On exit, \( t \) is overwritten by the local pieces of the reordered matrix \( T \), again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

\( q \)  
On exit, if \( \text{compq} = 'V' \), \( q \) has been postmultiplied by the global orthogonal transformation matrix which reorders \( t \); the leading \( m \) columns of \( q \) form an orthonormal basis for the specified invariant subspace.

If \( \text{compq} = 'N' \), \( q \) is not referenced.

\( \text{wr, wi} \)  
(global) array of size \( n \)  
The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix \( T \). The eigenvalues are in principle stored in the same order as on the diagonal of \( T \), with \( \text{wr}[i] = T(i+i+1) \) and, if \( T(i+i+1,:i+i+1) \) is a 2-by-2 diagonal block, \( \text{wi}[i-1] > 0 \) and \( \text{wi}[i] = -\text{wi}[i-1] \).

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.
The size of the specified invariant subspace.

\[ 0 \leq m \leq n. \]

\texttt{work[0]}  

On exit, if \( info = 0 \), \texttt{work[0]} returns the optimal \( lwork \).

\texttt{iwork[0]}  

On exit, if \( info = 0 \), \texttt{iwork[0]} returns the optimal \( liwork \).

\texttt{info}  

(general)  

\( = 0 \): successful exit

\( < 0 \): if \( info = -i \), the \( i \)-th argument had an illegal value. If the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value, then \( info = -(i*1000+j) \), if the \( i \)-th argument is a scalar and had an illegal value, then \( info = -i \).

\( > 0 \): here we have several possibilities

- Reordering of \( t \) failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); \( t \) may have been partially reordered, and \( wr \) and \( wi \) contain the eigenvalues in the same order as in \( t \).
  
  On exit, \( info = \{ \text{the index of } t \text{ where the swap failed (indexing starts at 1)} \} \).

- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.
  
  On exit, \( info = \{ \text{the index of } t \text{ where the swap failed}\} \).

- If \( info = n+1 \), there is no valid BLACS context (see the BLACS documentation for details).

\textbf{Application Notes}

The following alignment requirements must hold:

- \( mb_t = nb_t = mb_q = nb_q \)
- \( rsrct_t = rsrct_q \)
- \( csrct_t = csrct_q \)

All matrices must be blocked by a block factor larger than or equal to two (3). This is to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of \( t \) and \( q \), i.e., \( it = jt = iq = jq = 1 \) must hold. This is however no limitation since \texttt{p?lahqr} does not compute Schur forms of submatrices anyway.

Parallel execution recommendations:

- Use a square grid, if possible, for maximum performance. The block parameters in \texttt{para} should be kept well below the data distribution block size.
- In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

\textbf{See Also}

\textbf{Overview} for details of ScaLAPACK array descriptor structures and related notations.
p?trsen
Reorders the Schur factorization of a matrix and (optionally) computes the reciprocal condition numbers and invariant subspace for the selected cluster of eigenvalues.

Syntax

void pstrsen(char* job, char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, float* t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, float* q, MKL_INT* iq, MKL_INT* jq, MKL_INT* descq, float* wr, float* wi, MKL_INT* m, float* s, float* sep, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* info);

void pdtrsen(char* job, char* compq, MKL_INT* select, MKL_INT* para, MKL_INT* n, double* t, MKL_INT* it, MKL_INT* jt, MKL_INT* desct, double* q, MKL_INT* iq, MKL_INT* jq, MKL_INT* descq, double* wr, double* wi, MKL_INT* m, double* s, double* sep, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* info);

Include Files

• mkl_scalapack.h

Description

p?trsen reorders the real Schur factorization of a real matrix \( A = Q^T \Lambda Q \), so that a selected cluster of eigenvalues appears in the leading diagonal blocks of the upper quasi-triangular matrix \( T \), and the leading columns of \( Q \) form an orthonormal basis of the corresponding right invariant subspace. The reordering is performed by p?trord.

Optionally the function computes the reciprocal condition numbers of the cluster of eigenvalues and/or the invariant subspace.

\( T \) must be in Schur form (as returned by p?lahqr), that is, block upper triangular with 1-by-1 and 2-by-2 diagonal blocks.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

Input Parameters

job

Specifies whether condition numbers are required for the cluster of eigenvalues (s) or the invariant subspace (sep):
= 'N': no condition numbers are required;
= 'E': only the condition number for the cluster of eigenvalues is computed (s);
= 'V': only the condition number for the invariant subspace is computed
(sep);
= 'B': condition numbers for both the cluster and the invariant subspace are
computed (s and sep).

compq

(global )

= 'V': update the matrix q of Schur vectors;
= 'N': do not update q.

select

(global ) array of size n

select specifies the eigenvalues in the selected cluster. To select a real
eigenvalue \( w(j) \), \( select[j-1] \) must be set to a non-zero number. To select a
complex conjugate pair of eigenvalues \( w(j) \) and \( w(j+1) \), corresponding to a
2-by-2 diagonal block, either \( select[j-1] \) or \( select[j] \) or both must be set
to a non-zero number; a complex conjugate pair of eigenvalues must be
either both included in the cluster or both excluded.

para

(global )

Block parameters:

\[ para[0] \]
maximum number of concurrent computational
windows allowed in the algorithm; \( 0 < para[0] \leq \min(NPROW,NPCOL) \) must hold;

\[ para[1] \]
number of eigenvalues in each window; \( 0 < para[1] < para[2] \) must hold;

\[ para[2] \]
window size; \( para[1] < para[2] < mb_t \) must hold;

\[ para[3] \]
minimal percentage of flops required for
performing matrix-matrix multiplications instead
of pipelined orthogonal transformations; \( 0 \leq para[3] \leq 100 \) must hold;

\[ para[4] \]
width of block column slabs for row-wise
application of pipelined orthogonal
transformations in their factorized form; \( 0 < para[4] \leq mb_t \) must hold.

\[ para[5] \]
the maximum number of eigenvalues moved
together over a process border; in practice, this
will be approximately half of the cross border
window size \( 0 < para[5] \leq para[1] \) must hold;

n

(global )
The order of the globally distributed matrix \( t \). \( n \geq 0 \).

t

(local ) array of size \( lld_t \times LOC_c(n) \).
The local pieces of the global distributed upper quasi-triangular matrix \( T \), in
Schur form.

it, jt

(global )
The row and column index in the global matrix $T$ indicating the first column of $T$. $it = jt = 1$ must hold (see Application Notes).

$\text{desct}$

(global and local) array of size $\text{dlen}_\cdot$.

The array descriptor for the global distributed matrix $T$.

$q$

(local ) array of size $\text{lld}_q \times \text{LOC}_c(n)$.

On entry, if $\text{compq} = \text{'V'}$, the local pieces of the global distributed matrix $Q$ of Schur vectors.

If $\text{compq} = \text{'N'}$, $q$ is not referenced.

$iq, jq$

(global )

The column index in the global matrix $Q$ indicating the first column of $Q$. $iq = jq = 1$ must hold (see Application Notes).

$\text{descq}$

(global and local) array of size $\text{dlen}_\cdot$.

The array descriptor for the global distributed matrix $Q$.

$\text{work}$

(local workspace) array of size $\text{lwork}$

The size of the array $\text{work}$.

If $\text{lwork} = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $\text{work}$ array, returns this value as the first entry of the $\text{work}$ array, and no error message related to $\text{lwork}$ is issued by $\text{pxerbla}$.

$i\text{work}$

(local workspace) array of size $\text{liwork}$

(local )

The size of the array $\text{iwork}$.

If $\text{liwork} = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $\text{iwork}$ array, returns this value as the first entry of the $\text{iwork}$ array, and no error message related to $\text{liwork}$ is issued by $\text{pxerbla}$.

**OUTPUT Parameters**

$t$

$t$ is overwritten by the local pieces of the reordered matrix $T$, again in Schur form, with the selected eigenvalues in the globally leading diagonal blocks.

$q$

On exit, if $\text{compq} = \text{'V'}$, $q$ has been postmultiplied by the global orthogonal transformation matrix which reorders $t$; the leading $m$ columns of $q$ form an orthonormal basis for the specified invariant subspace.

If $\text{compq} = \text{'N'}$, $q$ is not referenced.

$wr, wi$

(global ) array of size $n$
The real and imaginary parts, respectively, of the reordered eigenvalues of the matrix $T$. The eigenvalues are in principle stored in the same order as on the diagonal of $T$, with $w_{r}[i] = T(i+1,i+1)$ and, if $T(i:i+1,i:i+1)$ is a 2-by-2 diagonal block, $w_{i}[i-1] > 0$ and $w_{i}[i] = -w_{i}[i-1]$.

Note also that if a complex eigenvalue is sufficiently ill-conditioned, then its value may differ significantly from its value before reordering.

$m$  
(global )

The size of the specified invariant subspace. $0 \leq m \leq n$.

$s$  
(global )

If $job = 'E'$ or 'B', $s$ is a lower bound on the reciprocal condition number for the selected cluster of eigenvalues. $s$ cannot underestimate the true reciprocal condition number by more than a factor of $\sqrt{n}$. If $m = 0$ or $n$, $s = 1$.

If $job = 'N'$ or 'V', $s$ is not referenced.

$sep$  
(global )

If $job = 'V'$ or 'B', $sep$ is the estimated reciprocal condition number of the specified invariant subspace. If $m = 0$ or $n$, $sep = \text{norm}(t)$.

If $job = 'N'$ or 'E', $sep$ is not referenced.

$work[0]$  
On exit, if $info = 0$, $work[0]$ returns the optimal $lwork$.

$iwork[0]$  
On exit, if $info = 0$, $iwork[0]$ returns the optimal $liwork$.

$info$  
(global )

= 0: successful exit

< 0: if $info = -i$, the $i$-th argument had an illegal value.

If the $i$-th argument is an array and the $j$-th entry, indexed $j-1$, had an illegal value, then $info = -(i*1000+j)$, if the $i$-th argument is a scalar and had an illegal value, then $info = -i$.

> 0: here we have several possibilities

- Reordering of $t$ failed because some eigenvalues are too close to separate (the problem is very ill-conditioned); $t$ may have been partially reordered, and $w_{r}$ and $w_{i}$ contain the eigenvalues in the same order as in $t$.

  On exit, $info = \{\text{the index of } t \text{ where the swap failed (indexing starts at 1)}\}$.

- A 2-by-2 block to be reordered split into two 1-by-1 blocks and the second block failed to swap with an adjacent block.

  On exit, $info = \{\text{the index of } t \text{ where the swap failed}\}$.

- If $info = n+1$, there is no valid BLACS context (see the BLACS documentation for details).

**Application Notes**

The following alignment requirements must hold:
All matrices must be blocked by a block factor larger than or equal to two (3). This to simplify reordering across processor borders in the presence of 2-by-2 blocks.

This algorithm cannot work on submatrices of \( t \) and \( q \), i.e., \( it = jt = iq = jq = 1 \) must hold. This is however no limitation since \( \text{plahqr} \) does not compute Schur forms of submatrices anyway.

For parallel execution, use a square grid, if possible, for maximum performance. The block parameters in \( \text{para} \) should be kept well below the data distribution block size.

In general, the parallel algorithm strives to perform as much work as possible without crossing the block borders on the main block diagonal.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

\textbf{p?trti2}

\textit{Computes the inverse of a triangular matrix (local unblocked algorithm).}

\textbf{Syntax}

\begin{verbatim}
void pstrti2 (char *uplo, char *diag, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pdtrti2 (char *uplo, char *diag, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pctrti2 (char *uplo, char *diag, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
void pztrti2 (char *uplo, char *diag, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *info);
\end{verbatim}

\textbf{Include Files}

- mkl_scalapack.h

\textbf{Description}

The \texttt{p?trti2} function computes the inverse of a real/complex upper or lower triangular block matrix sub \( (A) = A(ia:ia+n-1, ja:ja+n-1). \)

This matrix should be contained in one and only one process memory space (local operation).

\textbf{Input Parameters}

\begin{itemize}
  \item \texttt{uplo} (global)
    \text{Specifies whether the matrix sub \( (A) \) is upper or lower triangular.}
    \begin{itemize}
      \item = `'U'`: sub \( (A) \) is upper triangular
      \item = `'L'`: sub \( (A) \) is lower triangular.
    \end{itemize}

  \item \texttt{diag} (global)
    \text{Specifies whether or not the matrix \( A \) is unit triangular.}
    \begin{itemize}
      \item = `'N'`: sub \( (A) \) is non-unit triangular
    \end{itemize}
\end{itemize}
= 'U': sub (A) is unit triangular.

\( n \) (global)
The number of rows and columns to be operated on, i.e., the order of the distributed submatrix \( \text{sub}(A) \). \( n \geq 0 \).

\( a \) (local)
Pointer into the local memory to an array, \( \text{size llda} \ast \text{LOCc}(ja+n-1) \).
On entry, this array contains the local pieces of the triangular matrix \( \text{sub}(A) \).
If \( \text{uplo} = 'U' \), the leading \( n \)-by-\( n \) upper triangular part of the matrix \( \text{sub}(A) \) contains the upper triangular part of the matrix, and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced.
If \( \text{uplo} = 'L' \), the leading \( n \)-by-\( n \) lower triangular part of the matrix \( \text{sub}(A) \) contains the lower triangular part of the matrix, and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced. If \( \text{diag} = 'U' \), the diagonal elements of \( \text{sub}(A) \) are not referenced either and are assumed to be 1.

\( ia, ja \) (global)
The row and column indices in the global matrix \( A \) indicating the first row and the first column of the \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of size \( dlen_ \). The array descriptor for the distributed matrix \( A \).

**Output Parameters**

\( a \) On exit, the (triangular) inverse of the original matrix, in the same storage format.

\( \text{info} \)

\( = 0 \): successful exit
\(< 0 \): if the \( i \)-th argument is an array and the \( j \)-th entry, indexed \( j-1 \), had an illegal value,
then \( \text{info} = -(i*100+j) \),
if the \( i \)-th argument is a scalar and had an illegal value,
then \( \text{info} = -i \).

**See Also**

**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**?lahqr2**

*Updates the eigenvalues and Schur decomposition.*

**Syntax**

```c
void clahqr2 (const MKL_INT* wantt, const MKL_INT* wantz, const MKL_INT* n, const MKL_INT* ilo, const MKL_INT* ihi, MKL_Complex8* h, const MKL_INT* ldh, MKL_Complex8* w, const MKL_INT* iloz, const MKL_INT* ihiz, MKL_Complex8* z, const MKL_INT* ldz, MKL_INT* info);
```
void zlahqr2 (const MKL_INT* wantt, const MKL_INT* wantz, const MKL_INT* n, const MKL_INT* ilo, const MKL_INT* ihi, MKL_Complex16* h, const MKL_INT* ldh, MKL_Complex16* w, const MKL_INT* iloz, const MKL_INT* ihiz, MKL_Complex16* z, const MKL_INT* ldz, MKL_INT* info);

**Include Files**
- mkl_scalapack.h

**Description**
?lahqr2 is an auxiliary routine called by ?hseqr to update the eigenvalues and Schur decomposition already computed by ?hseqr, by dealing with the Hessenberg submatrix in rows and columns ilo to ihi. This version of ?lahqr (not the standard LAPACK version) uses a double-shift algorithm (like LAPACK’s ?lahqr). Unlike the standard LAPACK convention, this does not assume the subdiagonal is real, nor does it work to preserve this quality if given.

**Input Parameters**

**wantt**
- ≠ 0: the full Schur form T is required;
- = 0: only eigenvalues are required.

**wantz**
- ≠ 0: the matrix of Schur vectors Z is required;
- = 0: Schur vectors are not required.

**n**
The order of the matrix H. n >= 0.

**ilo, ihi**
It is assumed that the matrix H is upper triangular in rows and columns ihi +1 : n, and that matrix element H(ilo, ilo-1) = 0 (unless ilo = 1). ?lahqr works primarily with the Hessenberg submatrix in rows and columns ilo to ihi, but applies transformations to all of h if wantt is nonzero.
- 1 <= ilo <= max(1, ihi); ihi <= n.

**h**
Array, size ldh*n.
On entry, the upper Hessenberg matrix H.

**ldh**
The leading dimension of the array h. ldh >= max(1, n).

**iloz, ihiz**
Specify the rows of Z to which transformations must be applied if wantz ≠ 0.
- 1 <= iloz <= ilo; ihi <= ihiz <= n.

**z**
Array, size ldz*n.
If wantz ≠ 0, on entry z must contain the current matrix Z of transformations. If wantz = 0, z is not referenced.

**ldz**
The leading dimension of the array z. ldz >= max(1, n).
### Output Parameters

- **h**: On exit, if `wantt ≠ 0`, `h` is upper triangular in rows and columns `ilo` to `ihi`. If `wantt = 0`, the contents of `h` are unspecified on exit.

- **w**: Array, size (`n`)
  The computed eigenvalues `ilo` to `ihi` are stored in the corresponding elements of `w`. If `wantt ≠ 0`, the eigenvalues are stored in the same order as on the diagonal of the Schur form returned in `h`, with `w[i] = H(i, i)`.

- **z**: If `wantz ≠ 0`, on exit `z` has been updated; transformations are applied only to the submatrix `Z(ilo:ihiz, ilo:ihi)`. If `wantz = 0`, `z` is not referenced.

- **info**: 
  - `info = 0`: successful exit
  - `info > 0`: if `info = i`, `?lahqr` failed to compute all the eigenvalues `ilo` to `ihi` in a total of `30*(ihi-ilo+1)` iterations; elements `w[i:ihi - 1]` contain those eigenvalues which have been successfully computed.

---

**?lamsh**

*Sends multiple shifts through a small (single node) matrix to maximize the number of bulges that can be sent through.*

#### Syntax

```c
void slamsh (float *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk, float *h, const MKL_INT *ldh, const MKL_INT *n, const float *ulp);
void dlamsh (double *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk, double *h, const MKL_INT *ldh, const MKL_INT *n, const double *ulp);
void clamsh (MKL_Complex8 *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk, MKL_Complex8 *h, const MKL_INT *ldh, const MKL_INT *n, const float *ulp);
void zlamsh (MKL_Complex16 *s, const MKL_INT *lds, MKL_INT *nbulge, const MKL_INT *jblk, MKL_Complex16 *h, const MKL_INT *ldh, const MKL_INT *n, const double *ulp);
```

#### Include Files

- mkl_scalapack.h

#### Description

The `?lamsh` function sends multiple shifts through a small (single node) matrix to see how small consecutive subdiagonal elements are modified by subsequent shifts in an effort to maximize the number of bulges that can be sent through. The function should only be called when there are multiple shifts/bulges (`nbulge > 1`) and the first shift is starting in the middle of an unreduced Hessenberg matrix because of two or more small consecutive subdiagonal elements.

#### Input Parameters

- **s**: (local)
  Array of size `lds*2*jblk`. 
On entry, the matrix of shifts. Only the 2x2 diagonal of \( s \) is referenced. It is assumed that \( s \) has \( jblk \) double shifts (size 2).

\( lds \) (local)

On entry, the leading dimension of \( S \); unchanged on exit. \( 1 < nbulge \leq jblk \leq lds/2 \).

\( nbulge \) (local)

On entry, the number of bulges to send through \( h \) \((>1)\). \( nbulge \) should be less than the maximum determined \( (jblk) \). \( 1 < nbulge \leq jblk \leq lds/2 \).

\( jblk \) (local)

On entry, the number of double shifts determined for \( S \); unchanged on exit.

\( h \) (local)

Array of size \( ldh \times n \).

On entry, the local matrix to apply the shifts on. \( h \) should be aligned so that the starting row is 2.

\( ldh \) (local)

On entry, the leading dimension of \( H \); unchanged on exit.

\( n \) (local)

On entry, the size of \( H \). If all the bulges are expected to go through, \( n \) should be at least \( 4nbulge+2 \). Otherwise, \( nbulge \) may be reduced by this function.

\( ulp \) (local)

On entry, machine precision. Unchanged on exit.

**Output Parameters**

\( s \)

On exit, the data is rearranged in the best order for applying.

\( nbulge \)

On exit, the maximum number of bulges that can be sent through.

\( h \)

On exit, the data is destroyed.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lapst

_Sorts the numbers in increasing or decreasing order._

**Syntax**

```c
void slapst (const char* id, const MKL_INT* n, const float* d, MKL_INT* indx, MKL_INT* info);
void dlapst (const char* id, const MKL_INT* n, const double* d, MKL_INT* indx, MKL_INT* info);
```
Include Files

- mkl_scalapack.h

Description

?lapst is a modified version of the LAPACK routine ?lasrt. Define a permutation indx that sorts the numbers in d in increasing order (if id = 'I') or in decreasing order (if id = 'D').

Use Quick Sort, reverting to Insertion sort on arrays of size <= 20. Dimension of STACK limits n to about $2^{32}$.

Input Parameters

- id
  - = 'I': sort d in increasing order;
  - = 'D': sort d in decreasing order.
- n
  - The length of the array d.
- d
  - Array, size (n)
  - The array to be sorted.

Output Parameters

- indx
  - Array, size (n).
  - The permutation which sorts the array d.
- info
  - = 0: successful exit
  - < 0: if info = -i, the i-th argument had an illegal value

?laqr6

Performs a single small-bulge multi-shift QR sweep collecting the transformations.

Syntax

void slaqr6(char* job, MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nshfts, float* sr, float* si, float* h, MKL_INT* ldh, MKL_INT* iloz, MKL_INT* ihiz, float* z, MKL_INT* ldz, float* v, MKL_INT* ldv, float* u, MKL_INT* ldu, MKL_INT* nv, float* wv, MKL_INT* ldwv, MKL_INT* nh, float* wh, MKL_INT* ldwh);

void dlaqr6(char* job, MKL_INT* wantt, MKL_INT* wantz, MKL_INT* kacc22, MKL_INT* n, MKL_INT* ktop, MKL_INT* kbot, MKL_INT* nshfts, double* sr, double* si, double* h, MKL_INT* ldh, MKL_INT* iloz, MKL_INT* ihiz, double* z, MKL_INT* ldz, double* v, MKL_INT* ldv, double* u, MKL_INT* ldu, MKL_INT* nv, double* wv, MKL_INT* ldwv, MKL_INT* nh, double* wh, MKL_INT* ldwh);

Include Files

- mkl_scalapack.h
Description

This auxiliary function performs a single small-bulge multi-shift QR sweep, moving the chain of bulges from top to bottom in the submatrix $H(ktop:kbot, ktop:kbot)$, collecting the transformations in the matrix $V$ or accumulating the transformations in the matrix $Z$ (see below).

This is a modified version of ?laqr5 from LAPACK 3.1.

Input Parameters

$job$

Set the kind of job to do in ?laqr6, as follows:

- $job = 'I'$: Introduce and chase bulges in submatrix
- $job = 'C'$: Chase bulges from top to bottom of submatrix
- $job = 'O'$: Chase bulges off submatrix

$wantt$

$wantt$ is non-zero if the quasi-triangular Schur factor is being computed. $wantt$ is set to zero otherwise.

$wantz$

$wantz$ is non-zero if the orthogonal Schur factor is being computed. $wantz$ is set to zero otherwise.

$kacc22$

Specifies the computation mode of far-from-diagonal orthogonal updates.

- $kacc22 = 0$: ?laqr6 does not accumulate reflections and does not use matrix-matrix multiply to update far-from-diagonal matrix entries.
- $kacc22 = 1$: ?laqr6 accumulates reflections and uses matrix-matrix multiply to update the far-from-diagonal matrix entries.

$n$

$n$ is the order of the Hessenberg matrix $H$ upon which this function operates.

$ktop, kbot$

These are the first and last rows and columns of an isolated diagonal block upon which the QR sweep is to be applied. It is assumed without a check that either $ktop = 1$ or $H(ktop, ktop-1) = 0$ and either $kbot = n$ or $H(kbot+1, kbot) = 0$.

$nshfts$

$nshfts$ gives the number of simultaneous shifts. $nshfts$ must be positive and even.

$sr, si$

Array of size $nshfts$

$sr$ contains the real parts and $si$ contains the imaginary parts of the $nshfts$ shifts of origin that define the multi-shift QR sweep.

$h$

Array of size $ldh * n$

On input $h$ contains a Hessenberg matrix $H$.

$ldh$

$ldh$ is the leading dimension of $H$ just as declared in the calling function. $ldh \geq \max(1, n)$.

$iloz, ihiz$

Specify the rows of the matrix $Z$ to which transformations must be applied if $wantz$ is non-zero. $1 \leq iloz, ihiz \leq n$. 
**Array of size** $ldz \times ktop$

If $\text{wantz}$ is non-zero, then the QR sweep orthogonal similarity transformation is accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$, stored in the array $z$, from the right.

If $\text{wantz}$ equals zero, then $z$ is unreferenced.

$ldz$

$ldz$ is the leading dimension of $z$ just as declared in the calling function. $ldz \geq n$.

$v$

(workspace) array of size $ldv \times nshfts/2$

$ldv$

$ldv$ is the leading dimension of $v$ as declared in the calling function. $ldv \geq 3$.

$u$

(workspace) array of size $ldu \times (3 \times nshfts - 3)$

$ldu$

$ldu$ is the leading dimension of $u$ just as declared in the calling function. $ldu \geq 3 \times nshfts - 3$.

$nh$

$nh$ is the number of columns in array $wh$ available for workspace. $nh \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

$wh$

(workspace) array of size $ldwh \times nh$

$ldwh$

Leading dimension of $wh$ just as declared in the calling function. $ldwh \geq 3 \times nshfts - 3$.

$nv$

$nv$ is the number of rows in $wv$ available for workspace. $nv \geq 1$ is required for usage of this workspace, otherwise the updates of the far-from-diagonal elements will be updated without level 3 BLAS.

$wv$

(workspace) array of size $ldwv \times 3 \times nshfts$

$ldwv$

Scalar

$ldwv$ is the leading dimension of $wv$ as declared in the in the calling function. $ldwv \geq nv$.

**OUTPUT Parameters**

$h$

A multi-shift QR sweep with shifts $si[j]+i*si[j]$ is applied to the isolated diagonal block in matrix rows and columns $ktop$ through $kbot$.

$z$

If $\text{wantz}$ is non-zero, then the QR sweep orthogonal/unitary similarity transformation is accumulated into the matrix $Z(iloz:ihiz,kbot:ktop)$ from the right.

If $\text{wantz}$ equals zero, then $z$ is unreferenced.

**Application Notes**

**Notes**

Based on contributions by Karen Braman and Ralph Byers, Department of Mathematics, University of Kansas, USA Robert Granat, Department of Computing Science and HPC2N, Umea University, Sweden

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
?lar1va
Computes scaled eigenvector corresponding to given eigenvalue.

Syntax
void slar1va (MKL_INT* n, MKL_INT* b1, MKL_INT* bn, float* lambda, float* d, float* l, float* ld, float* lld, float* pivmin, float* gaptol, float* z, MKL_INT* wantnc, MKL_INT* negcnt, float* ztz, float* mingma, MKL_INT* r, MKL_INT* isuppz, float* nrminv, float* resid, float* rqcorr, float* work);
void dlar1va (MKL_INT* n, MKL_INT* b1, MKL_INT* bn, double* lambda, double* d, double* l, double* ld, double* lld, double* pivmin, double* gaptol, double* z, MKL_INT* wantnc, MKL_INT* negcnt, double* ztz, double* mingma, MKL_INT* r, MKL_INT* isuppz, double* nrminv, double* resid, double* rqcorr, double* work);

Include Files
• mkl_scalapack.h

Description
?lar1va computes the (scaled) r-th column of the inverse of the submatrix in rows b1 through bn of the tridiagonal matrix \( LDL^T - \lambda I \). When \( \lambda \) is close to an eigenvalue, the computed vector is an accurate eigenvector. Usually, \( r \) corresponds to the index where the eigenvector is largest in magnitude. The following steps accomplish this computation:

1. Stationary qd transform, \( LDL^T - \lambda I = L_1 D_1 L_1^T \).
2. Progressive qd transform, \( LDL^T - \lambda I = U D U^T \).
3. Computation of the diagonal elements of the inverse of \( LDL^T - \lambda I \) by combining the above transforms, and choosing \( r \) as the index where the diagonal of the inverse is (one of the) largest in magnitude.
4. Computation of the (scaled) r-th column of the inverse using the twisted factorization obtained by combining the top part of the stationary and the bottom part of the progressive transform.

Input Parameters

n
The order of the matrix \( LDL^T \).

b1
First index of the submatrix of \( LDL^T \).

bn
Last index of the submatrix of \( LDL^T \).

lambda
The shift \( \lambda \). In order to compute an accurate eigenvector, \( \lambda \) should be a good approximation to an eigenvalue of \( LDL^T \).

l
Array of size \( n-1 \)

The \( (n-1) \) subdiagonal elements of the unit bidiagonal matrix \( L \), in elements 0 to \( n-2 \).

d
Array of size \( n \)

The \( n \) diagonal elements of the diagonal matrix \( D \).

ld
Array of size \( n-1 \)

The \( n-1 \) elements \( l[i]*d[i], i=0,\ldots,n-2 \).

lld
Array of size \( n-1 \)
The \( n-1 \) elements \( l[i,i] \cdot d[i], i=0, \ldots, n-2 \).

**pivmin**
The minimum pivot in the Sturm sequence.

**gaptol**
Tolerance that indicates when eigenvector entries are negligible with respect to their contribution to the residual.

**z**
Array of size \( n \)

On input, all entries of \( z \) must be set to 0.

**wantnc**
Specifies whether \( \text{negcnt} \) has to be computed.

**r**
The twist index for the twisted factorization used to compute \( z \).

On input, \( 0 \leq r \leq n \). If \( r \) is input as 0, \( r \) is set to the index where \((LDL^T - \sigma I)^{-1}\) is largest in magnitude. If \( 1 \leq r \leq n \), \( r \) is unchanged.

Ideally, \( r \) designates the position of the maximum entry in the eigenvector.

**work**
(Workspace) array of size \( 4 \times n \)

**OUTPUT Parameters**

**z**
On output, \( z \) contains the (scaled) \( r \)-th column of the inverse. The scaling is such that \( z[r-1] \) equals 1.

**negcnt**
If \( \text{wantncis non-zero} \) then \( \text{negcnt} = \text{the number of pivots} < \text{pivmin} \) in the matrix factorization \( LDL^T \), and \( \text{negcnt} = -1 \) otherwise.

**ztz**
The square of the 2-norm of \( z \).

**mingma**
The reciprocal of the largest (in magnitude) diagonal element of the inverse of \( LDL^T - \sigma I \).

**r**
On output, \( r \) contains the twist index used to compute \( z \).

**isuppz**
array of size 2

The support of the vector in \( z \), i.e., the vector \( z \) is non-zero only in elements \( \text{isuppz}[0] \) and \( \text{isuppz}[1] \).

**nrminv**
\( nrminv = 1/\text{SQRT}(ztz) \)

**resid**
The residual of the FP vector.

\( resid = \text{ABS}(mingma)/\text{SQRT}(ztz) \)

**rqcorr**
The Rayleigh Quotient correction to \( \lambda \).

**See Also**
**Overview** for details of ScaLAPACK array descriptor structures and related notations.

**?laref**
Applies Householder reflectors to matrices on their rows or columns.
Syntax

void slaref (const char* type, float* a, const MKL_INT* lda, const MKL_INT* wantz, float* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* lihiz, const float* vecs, float* v2, float* v3, float* t1, float* t2, float* t3);

void dlaref (const char* type, double* a, const MKL_INT* lda, const MKL_INT* wantz, double* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* lihiz, const double* vecs, double* v2, double* v3, double* t1, double* t2, double* t3);

void claref (const char* type, MKL_Complex8* a, const MKL_INT* lda, const MKL_INT* wantz, MKL_Complex8* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* lihiz, const MKL_Complex8* vecs, MKL_Complex8* v2, MKL_Complex8* v3, MKL_Complex8* t1, MKL_Complex8* t2, MKL_Complex8* t3);

void zlaref (const char* type, MKL_Complex16* a, const MKL_INT* lda, const MKL_INT* wantz, MKL_Complex16* z, const MKL_INT* ldz, const MKL_INT* block, MKL_INT* irow1, MKL_INT* icoll, const MKL_INT* istart, const MKL_INT* istop, const MKL_INT* itmp1, const MKL_INT* itmp2, const MKL_INT* liloz, const MKL_INT* lihiz, const MKL_Complex16* vecs, MKL_Complex16* v2, MKL_Complex16* v3, MKL_Complex16* t1, MKL_Complex16* t2, MKL_Complex16* t3);

Include Files

• mkl_scalapack.h

Description

?laref applies one or several Householder reflectors of size 3 to one or two matrices (if column is specified) on either their rows or columns.

Input Parameters

type
(local)

If 'R': Apply reflectors to the rows of the matrix (apply from left)
Otherwise: Apply reflectors to the columns of the matrix
Unchanged on exit.

a
(local)

Array, lda*LOCc(ja+n-1)

On entry, the matrix to receive the reflections.

lda
(local)

On entry, the leading dimension of a.

Unchanged on exit.

wantz
(local)
If \( \text{want} z \neq 0 \), then apply any column reflections to \( z \) as well.

If \( \text{want} z = 0 \), then do no additional work on \( z \).

\( z \)

(local)

Array, \( ldz * ncols \), where the value \( ncols \) depends on other arguments. If \( \text{want} z \text{want} z \neq 0 \) and \( \text{type} \neq 'R' \) then \( ncols = icoil + 3*(lihiz - liloz + 1) \). Otherwise, \( ncols \) is unused.

On entry, the second matrix to receive column reflections.

This is changed only if \( \text{want} z \) is set.

\( ldz \)

(local)

On entry, the leading dimension of \( z \).

Unchanged on exit.

\( \text{block} \)

(local)

If nonzero, then apply several reflectors at once and read their data from the \( \text{vecs} \) array.

If zero, apply the single reflector given by \( v2, v3, t1, t2, \) and \( t3 \).

\( irow1 \)

(local)

On entry, the local row element of \( a \).

\( icol1 \)

(local)

On entry, the local column element of \( a \).

\( \text{istart} \)

(local)

Specifies the "number" of the first reflector. This is used as an index into \( \text{vecs} \) if \( \text{block} \) is set. \( \text{istart} \) is ignored if \( \text{block} \) is zero.

\( \text{istop} \)

(local)

Specifies the "number" of the last reflector. This is used as an index into \( \text{vecs} \) if \( \text{block} \) is set. \( \text{istop} \) is ignored if \( \text{block} \) is zero.

\( \text{itmp1} \)

(local)

Starting range into \( a \). For rows, this is the local first column. For columns, this is the local first row.

\( \text{itmp2} \)

(local)

Ending range into \( a \). For rows, this is the local last column. For columns, this is the local last row.

\( \text{liloz, lihiz} \)

(local)

These serve the same purpose as \( \text{itmp1, itmp2} \) but for \( z \) when \( \text{want} z \) is set.

\( \text{vecs} \)

(local)
Array of size $3\times N$ (matrix size)
This holds the size 3 reflectors one after another and this is only accessed when block is nonzero

$v_2, v_3, t_1, t_2, t_3$ (local)
This holds information on a single size 3 Householder reflector and is read when block is zero, and overwritten when block is nonzero

**Output Parameters**

*a*
The updated matrix on exit.

*z*
This is changed only if wantz is set.

*irow1*
Undefined on output.

*icol1*
Undefined on output.

$v_2, v_3, t_1, t_2, t_3$ Overwritten when block is nonzero.

---

**slarrb2**
*Provides limited bisection to locate eigenvalues for more accuracy.*

**Syntax**

```c
void slarrb2(MKL_INT* n, float* d, float* lld, MKL_INT* ifirst, MKL_INT* ilast, float* rtol1, float* rtol2, MKL_INT* offset, float* w, float* wgap, float* werr, float* work, MKL_INT* iwork, float* pivmin, float* lgpvmn, float* lgspdm, MKL_INT* twist, MKL_INT* info);
```

```c
void dlarrb2(MKL_INT* n, double* d, double* lld, MKL_INT* ifirst, MKL_INT* ilast, double* rtol1, double* rtol2, MKL_INT* offset, double* w, double* wgap, double* werr, double* work, MKL_INT* iwork, double* pivmin, double* lgpvmn, double* lgspdm, MKL_INT* twist, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

Given the relatively robust representation (RRR) $LDL^T$, ?larrb2 does "limited" bisection to refine the eigenvalues of $LDL^T$ with indices in a given range to more accuracy. Initial guesses for these eigenvalues are input in $w$, the corresponding estimate of the error in these guesses and their gaps are input in $werr$ and $wgap$, respectively. During bisection, intervals $[left, right]$ are maintained by storing their mid-points and semi-widths in the arrays $w$ and $werr$ respectively. The range of indices is specified by the $ifirst$, $ilast$, and $offset$ parameters, as explained in Input Parameters.
NOTE
There are very few minor differences between larrb from LAPACK and this current function ?larrb2. The most important reason for creating this nearly identical copy is profiling: in the ScaLAPACK MRRR algorithm, eigenvalue computation using ?larrb2 is used for refinement in the construction of the representation tree, as opposed to the initial computation of the eigenvalues for the root RRR which uses ?larrb. When profiling, this allows an easy quantification of refinement work vs. computing eigenvalues of the root.

Input Parameters

\( n \)

The order of the matrix.

\( d \)

Array of size \( n \).

The \( n \) diagonal elements of the diagonal matrix \( D \).

\( lld \)

Array of size \( n-1 \).

The \((n-1)\) elements \( l_{i+1} \cdot l_{i+1} \cdot d[i] \), \( i=0, ..., n-2 \).

\( ifirst \)

The index of the first eigenvalue to be computed.

\( ilast \)

The index of the last eigenvalue to be computed.

\( rtol1, rtol2 \)

Tolerance for the convergence of the bisection intervals.

An interval \([left, right]\) has converged if \( right - left < \max (rtol1 \cdot \text{gap}, rtol2 \cdot \max (|left|, |right|)) \) where \( \text{gap} \) is the (estimated) distance to the nearest eigenvalue.

\( offset \)

Offset for the arrays \( w, \text{wgap and werr} \), i.e., the elements indexed \( ifirst - offset - 1 \) through \( ilast - offset - 1 \) of these arrays are to be used.

\( w \)

Array of size \( n \).

On input, \( w[ifirst - offset - 1] \) through \( w[ilast - offset - 1] \) are estimates of the eigenvalues of \( LDL^T \) indexed \( ifirst \) through \( ilast \).

\( wgap \)

Array of size \( n-1 \).

On input, the (estimated) gaps between consecutive eigenvalues of \( LDL^T \), i.e., \( wgap[I - offset - 1] \) is the gap between eigenvalues \( I \) and \( I + 1 \). Note that if \( ifirst = ilast \) then \( wgap[ifirst - offset - 1] \) must be set to zero.

\( werr \)

Array of size \( n \).

On input, \( werr[ifirst - offset - 1] \) through \( werr[ilast - offset - 1] \) are the errors in the estimates of the corresponding elements in \( w \).

\( work \)

(workspace) array of size \( 4 \cdot n \).

Workspace.

\( iwork \)

(workspace) array of size \( 2 \cdot n \).

Workspace.

\( pivmin \)

The minimum pivot in the Sturm sequence.
Logarithm of $\text{pivmin}$, precomputed.

Logarithm of the spectral diameter, precomputed.

The twist index for the twisted factorization that is used for the $\text{negcount}$.

- $\text{twist} = n$: Compute $\text{negcount}$ from $\text{LDL}^T - \lambda I = L_d D_d L_d^T$
- $\text{twist} = 1$: Compute $\text{negcount}$ from $\text{LDL}^T - \lambda I = U_d D_d U_d^T$
- $\text{twist} = r, 1 < r < n$: Compute $\text{negcount}$ from $\text{LDL}^T - \lambda I = N_r \Delta_r N_r^T$

**OUTPUT Parameters**

- $w$: On output, the eigenvalue estimates in $w$ are refined.
- $\text{wgap}$: On output, the eigenvalue gaps in $\text{wgap}$ are refined.
- $\text{werr}$: On output, the errors in $\text{werr}$ are refined.
- $\text{info}$: Error flag.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larrd2

*Computes the eigenvalues of a symmetric tridiagonal matrix to suitable accuracy.*

**Syntax**

```c
#include <mkl_scalapack.h>

void slarrd2(char* range, char* order, MKL_INT* n, float* vl, float* vu, MKL_INT* il,
MKL_INT* iu, float* gers, float* reltol, float* d, float* e, float* e2, float* pivmin,
MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, float* w, float* werr, float* wl, float* wu,
MKL_INT* iblock, MKL_INT* indexw, float* work, MKL_INT* iwork, MKL_INT* doli,
MKL_INT* dou, MKL_INT* info);

void dlarrd2(char* range, char* order, MKL_INT* n, double* vl, double* vu, MKL_INT* il,
MKL_INT* iu, double* gers, double* reltol, double* d, double* e, double* e2, double* pivmin,
MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, double* w, double* werr, double* wl,
double* wu, MKL_INT* iblock, MKL_INT* indexw, double* work, MKL_INT* iwork, MKL_INT* doli,
MKL_INT* dou, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

?larrd2 computes the eigenvalues of a symmetric tridiagonal matrix $T$ to limited initial accuracy. This is an auxiliary code to be called from larre2a.

?larrd2 has been created using the LAPACK code larrd which itself stems from stebz. The motivation for creating ?larrd2 is efficiency: When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, ?larrd2 can skip over blocks which contain none of the eigenvalues from DOL to DOU for which the processor responsible. In extreme cases (such as large matrices consisting of many blocks of small size like $2 \times 2$), the gain can be substantial.
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for
optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and
SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or
effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-
dependent optimizations in this product are intended for use with Intel microprocessors. Certain
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to
the applicable product User and Reference Guides for more information regarding the specific instruction
sets covered by this notice.

Notice revision #20110804

Input Parameters

**range**

= 'A': ("All") all eigenvalues will be found.
= 'V': ("Value") all eigenvalues in the half-open interval (vl, vu] will be
found.
= 'I': ("Index") eigenvalues of the entire matrix with the indices in a given
range will be found.

**order**

= 'B': ("By Block") the eigenvalues will be grouped by split-off block (see
iblock, isplit) and ordered from smallest to largest within the block.
= 'E': ("Entire matrix") the eigenvalues for the entire matrix will be ordered
from smallest to largest.

**n**

The order of the tridiagonal matrix \(T\). \(n \geq 0\).

**vl, vu**

If range='V', the lower and upper bounds of the interval to be searched for
eigenvalues. Eigenvalues less than or equal to \(vl\), or greater than \(vu\), will
not be returned. \(vl < vu\).

Not referenced if range = 'A' or 'I'.

**il, iu**

If range='I', the indices (in ascending order) of the smallest eigenvalue, to
be returned in \(w[il-1]\), and largest eigenvalue, to be returned in \(w[iu-1]\).
\(1 \leq il \leq iu \leq n\), if \(n > 0\); \(il = 1\) and \(iu = 0\) if \(n = 0\).

Not referenced if range = 'A' or 'V'.

**gers**

Array of size 2*\(n\)
The \(n\) Gerschgorin intervals (the \(i\)-th Gerschgorin interval is \(gers[2*i-2],
gers[2*i-1])\).

**reltol**

The minimum relative width of an interval. When an interval is narrower
than reltol times the larger (in magnitude) endpoint, then it is considered
to be sufficiently small, i.e., converged. Note: this should always be at least
radix*machine epsilon.

**d**

Array of size \(n\)
The \(n\) diagonal elements of the tridiagonal matrix \(T\).

**e**

Array of size \(n-1\)
The \((n-1)\) off-diagonal elements of the tridiagonal matrix \(T\).
e2

Array of size \( n-1 \)
The \((n-1)\) squared off-diagonal elements of the tridiagonal matrix \( T \).

pivmin

The minimum pivot allowed in the sturm sequence for \( T \).

nsplit

The number of diagonal blocks in the matrix \( T \).
\[ 1 \leq \text{nsplit} \leq n. \]

isplit

Array of size \( n \)
The splitting points, at which \( T \) breaks up into submatrices.
The first submatrix consists of rows/columns 1 to \( \text{isplit}[0] \), the second of rows/columns \( \text{isplit}[0]+1 \) through \( \text{isplit}[1] \), etc., and the \( \text{nsplit} \)-th submatrix consists of rows/columns \( \text{isplit}[\text{nsplit}-2]+1 \) through \( \text{isplit}[\text{nsplit}-1]=n \).
(Only the first \( \text{nsplit} \) elements will actually be used, but since the user cannot know a priori what value \( \text{nsplit} \) will have, \( n \) words must be reserved for \( \text{isplit} \).

work

(workspace) Array of size \( 4*n \)

iwork

(workspace) Array of size \( 3*n \)

Specifying an index range \( \text{dol:dou} \) allows the user to work on only a selected part of the representation tree.
Otherwise, the setting \( \text{dol}=1, \text{dou}=n \) should be applied.
Note that \( \text{dol} \) and \( \text{dou} \) refer to the order in which the eigenvalues are stored in \( W \).

OUTPUT Parameters

\( m \)

The actual number of eigenvalues found. \( 0 \leq m \leq n \).
(See also the description of \( \text{info}=2,3 \).)

\( w \)

Array of size \( n \)
On exit, the first \( m \) elements of \( w \) will contain the eigenvalue approximations. \( \text{?larrd2} \) computes an interval \( I_j = (a_j, b_j) \) that includes eigenvalue \( j \). The eigenvalue approximation is given as the interval midpoint \( w[j-1] = (a_j + b_j)/2 \). The corresponding error is bounded by \( \text{werr}[j-1] = \text{abs}(a_j - b_j)/2 \).

\( \text{werr} \)

Array of size \( n \)
The error bound on the corresponding eigenvalue approximation in \( w \).

\( \text{wl, wu} \)

The interval \( (\text{wl, wu}) \) contains all the wanted eigenvalues.
If \( \text{range}='V' \), then \( \text{wl} = \text{vl} \) and \( \text{wu} = \text{vu} \).
If \( \text{range}='A' \), then \( \text{wl} \) and \( \text{wu} \) are the global Gerschgorin bounds on the spectrum.
If \( \text{range}='I' \), then \( \text{wl} \) and \( \text{wu} \) are computed by \( \text{SLAEBZ} \) from the index range specified.
iblock

Array of size \( n \)

At each row/column \( j \) where \( e[j-1] \) is zero or small, the matrix \( T \) is considered to split into a block diagonal matrix. On exit, if \( \text{info} = 0 \), \( \text{iblock}[i] \) specifies to which block (from 0 to the number of blocks minus one) the eigenvalue \( w[i] \) belongs. (?larrd2 may use the remaining \( n-m \) elements as workspace.)

indexw

Array of size \( n \)

The indices of the eigenvalues within each block (submatrix); for example, \( \text{indexw}[i] = j \) and \( \text{iblock}[i] = k \) imply that the \((i+1)\)-th eigenvalue \( w[i] \) is the \( j \)-th eigenvalue in block \( k \).

info

\( = 0: \) successful exit
\( < 0: \) if \( \text{info} = -i \), the \( i \)-th argument had an illegal value
\( > 0: \) some or all of the eigenvalues failed to converge or were not computed:
  - \( =1 \) or \( 3: \) Bisection failed to converge for some eigenvalues; these eigenvalues are flagged by a negative block number. The effect is that the eigenvalues may not be as accurate as the absolute and relative tolerances.
  - \( =2 \) or \( 3: \) \( \text{range} = 'I' \) only: Not all of the eigenvalues \( il:iu \) were found.
  - \( =4: \) \( \text{range} = 'I', \) and the Gershgorin interval initially used was too small. No eigenvalues were computed.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?larr2

Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```c
void slarr2(char* range, MKL_INT* n, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, 
float* d, float* e, float* e2, float* rto1, float* rto2, float* spltol, MKL_INT* 
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, float* w, float* werr, 
float* wgap, MKL_INT* iblock, MKL_INT* indexw, float* gers, float* pivmin, float* work, 
MKL_INT* iwork, MKL_INT* info);

void dlarr2(char* range, MKL_INT* n, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, 
double* d, double* e, double* e2, double* rto1, double* rto2, double* spltol, MKL_INT* 
nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, double* w, double* 
werr, double* wgap, MKL_INT* iblock, MKL_INT* indexw, double* gers, double* pivmin, 
double* work, MKL_INT* iwork, MKL_INT* info);
```

Include Files

- `mkl_scalapack.h`
Description
To find the desired eigenvalues of a given real symmetric tridiagonal matrix $T$, \texttt{?larre2} sets, via \texttt{?larra}, "small" off-diagonal elements to zero. For each block $T_i$, it finds

- a suitable shift at one end of the block's spectrum,
- the root RRR, $T_i - \sigma_i L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

The representations and eigenvalues found are then returned to \texttt{?stegr2} to compute the eigenvectors $T$.

\texttt{?larre2} is more suitable for parallel computation than the original LAPACK code for computing the root RRR and its eigenvalues. When computing eigenvalues in parallel and the input tridiagonal matrix splits into blocks, \texttt{?larre2} can skip over blocks which contain none of the eigenvalues from $dol$ to $dou$ for which the processor is responsible. In extreme cases (such as large matrices consisting of many blocks of small size, e.g. 2x2), the gain can be substantial.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

\texttt{range}

- 'A': ("All") all eigenvalues will be found.
- 'V': ("Value") all eigenvalues in the half-open interval $(vl, vu]$ will be found.
- 'I': ("Index") eigenvalues of the entire matrix with the indices in a given range will be found.

\texttt{n}

The order of the matrix. $n > 0$.

\texttt{vl, vu}

If \texttt{range}='V', the lower and upper bounds for the eigenvalues.

Eigenvalues less than or equal to $vl$, or greater than $vu$, will not be returned. $vl < vu$.

\texttt{il, iu}

If \texttt{range}='I', the indices (in ascending order) of the smallest eigenvalue, to be returned in $w[il-1]$, and largest eigenvalue, to be returned in $w[iu-1]$. $1 \leq il \leq iu \leq n$.

\texttt{d}

Array of size $n$

The $n$ diagonal elements of the tridiagonal matrix $T$.

\texttt{e}

Array of size $n$

The first $(n-1)$ entries contain the subdiagonal elements of the tridiagonal matrix $T$; $e[n-1]$ need not be set.
The first \((n-1)\) entries contain the squares of the subdiagonal elements of the tridiagonal matrix \(T\); \(e_2[n-1]\) need not be set.

Parameters for bisection.

An interval \([\text{left}, \text{right}]\) has converged if \(\text{right-left}\leq \max(\text{rtol1}\times\text{gap}, \text{rtol2}\times\max(|\text{left}|,|\text{right}|))\)

The threshold for splitting.

Specifying an index range \(\text{dol: dou}\) allows the user to work on only a selected part of the representation tree. Otherwise, the setting \(\text{dol}=1, \text{dou}=n\) should be applied.

Note that \(\text{dol}\) and \(\text{dou}\) refer to the order in which the eigenvalues are stored in \(w\).

Workspace array of size \(6\times n\)

Workspace array of size \(5\times n\)

If \(\text{range}='I'\) or \(='A'\), \(\text{larrre2}\) contains bounds on the desired part of the spectrum.

The \(n\) diagonal elements of the diagonal matrices \(D_i\).

The entries \(e[\text{isplit}][i]\), \(0\leq\text{isplit}\), contain the base points \(\sigma_{i+1}\) on output.

The entries \(e_2[\text{isplit}][i]\), \(0\leq\text{isplit}\), are set to zero.

The number of blocks \(T\) splits into. \(1\leq\text{nsplit}\leq n\).

Array of size \(n\)

The splitting points, at which \(T\) breaks up into blocks.

The first block consists of rows/columns 1 to \(\text{isplit}[0]\), the second of rows/columns \(\text{isplit}[0]+1\) through \(\text{isplit}[1]\), etc., and the \(\text{nsplit}\)-th block consists of rows/columns \(\text{isplit}[\text{nsplit}-2]+1\) through \(\text{isplit}[\text{nsplit}-1]=n\).

The total number of eigenvalues (of all \(L_iD_iL_i^\top\)) found.

Array of size \(n\)

The first \(m\) elements contain the eigenvalues. The eigenvalues of each of the blocks, \(L_iD_iL_i^\top\), are sorted in ascending order (\(\text{larrre2}\) may use the remaining \(n-m\) elements as workspace).

Note that immediately after exiting this function, only the eigenvalues in \(w\) with indices in range \(\text{dol}-1:\text{dou}-1\) might rely on this processor when the eigenvalue computation is done in parallel.
The error bound on the corresponding eigenvalue in $w$.

Note that immediately after exiting this function, only the uncertainties in $werr$ with indices in range $dol-1:dou-1$ might rely on this processor when the eigenvalue computation is done in parallel.

$w_{gap}$

Array of size $n$

The separation from the right neighbor eigenvalue in $w$.

The gap is only with respect to the eigenvalues of the same block as each block has its own representation tree.

Exception: at the right end of a block we store the left gap

Note that immediately after exiting this function, only the gaps in $w_{gap}$ with indices in range $dol-1:dou-1$ might rely on this processor when the eigenvalue computation is done in parallel.

$iblock$

Array of size $n$

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in $w$; $iblock[i]=1$ if eigenvalue $w[i]$ belongs to the first block from the top, $iblock[i]=2$ if $w[i]$ belongs to the second block, and so on.

$indexw$

Array of size $n$

The indices of the eigenvalues within each block (submatrix); for example, $indexw[i]=10$ and $iblock[i]=2$ imply that the $(i+1)$-th eigenvalue $w[i]$ is the 10th eigenvalue in block 2.

$gers$

Array of size $2*n$

The $n$ Gerschgorin intervals (the $i$-th Gerschgorin interval is $(gers[2*i-2], gers[2*i-1])$).

$pivmin$

The minimum pivot in the sturm sequence for $T$.

$info$

- $= 0$: successful exit
- $> 0$: A problem occurred in $?larrre2$.
- $< 0$: One of the called functions signaled an internal problem.

Needs inspection of the corresponding parameter $info$ for further information.

- $=-1$: Problem in $?larrd$.
- $=-2$: Not enough internal iterations to find the base representation.
- $=-4$: Problem in $?larrb$ when preforming bisection on the desired part of the spectrum.
- $=-5$: Problem in $?lasq2$
- $=-6$: Problem in $?lasq2$

See Also

Overview for details of ScALAPACK array descriptor structures and related notations.
?larre2a
Given a tridiagonal matrix, sets small off-diagonal elements to zero and for each unreduced block, finds base representations and eigenvalues.

Syntax

```c
void slarre2a(char* range, MKL_INT* n, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, float* d, float* e, float* e2, float* rtol1, float* rtol2, float* spltol, MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, float* w, float* werr, float* wgap, MKL_INT* iblock, MKL_INT* indexw, float* gers, float* sdiam, float* pivmin, float* work, MKL_INT* iwork, float* minrgp, MKL_INT* info);

void dlarre2a(char* range, MKL_INT* n, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, double* d, double* e, double* e2, double* rtol1, double* rtol2, double* spltol, MKL_INT* nsplit, MKL_INT* isplit, MKL_INT* m, MKL_INT* Dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, double* w, double* werr, double* wgap, MKL_INT* iblock, MKL_INT* indexw, double* gers, double* sdiam, double* pivmin, double* work, MKL_INT* iwork, double* minrgp, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

Description

To find the desired eigenvalues of a given real symmetric tridiagonal matrix $T$, ?larre2a sets any "small" off-diagonal elements to zero, and for each unreduced block $T_i$, it finds

- a suitable shift at one end of the block's spectrum,
- the base representation, $T_i - \sigma_i I = L_i D_i L_i^T$, and
- eigenvalues of each $L_i D_i L_i^T$.

NOTE
The algorithm obtains a crude picture of all the wanted eigenvalues (as selected by range). However, to reduce work and improve scalability, only the eigenvalues $dol$ to $dou$ are refined. Furthermore, if the matrix splits into blocks, RRRs for blocks that do not contain eigenvalues from $dol$ to $dou$ are skipped. The DQDS algorithm (function ?lasq2) is not used, unlike in the sequential case. Instead, eigenvalues are computed in parallel to some figures using bisection.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Input Parameters

range

- 'A': ("All") all eigenvalues will be found.
- 'V': ("Value") all eigenvalues in the half-open interval \((vl, vu]\) will be found.
- 'I': ("Index") eigenvalues of the entire matrix with the indices in a given range will be found.

n

The order of the matrix. \(n > 0\).

vl, vu

If \(range='V'\), the lower and upper bounds for the eigenvalues. Eigenvalues less than or equal to \(vl\), or greater than \(vu\), will not be returned. \(vl < vu\).

If \(range='I'\) or \(='A'\), \(?larre2a\) computes bounds on the desired part of the spectrum.

il, iu

If \(range='I'\), the indices (in ascending order) of the smallest eigenvalue, to be returned in \(w[il-1]\), and largest eigenvalue, to be returned in \(w[iu-1]\). \(1 \leq il \leq iu \leq n\).

d

Array of size \(n\)

On entry, the \(n\) diagonal elements of the tridiagonal matrix \(T\).

e

Array of size \(n\)

The first \((n-1)\) entries contain the subdiagonal elements of the tridiagonal matrix \(T\); \(e[n-1]\) need not be set.

e2

Array of size \(n\)

The first \((n-1)\) entries contain the squares of the subdiagonal elements of the tridiagonal matrix \(T\); \(e2[n-1]\) need not be set.

rtol1, rtol2

Parameters for bisection.

An interval \([left, right]\) has converged if \(right - left < max(\ rtol1*gap, \ rtol2*max(|left|,|right|))\)

spltol

The threshold for splitting.

dol, dou

If the user wants to work on only a selected part of the representation tree, he can specify an index range \(dol:dou\).

Otherwise, the setting \(dol=1, dou=n\) should be applied.

Note that \(dol\) and \(dou\) refer to the order in which the eigenvalues are stored in \(w\).

work

Workspace array of size \(6*n\)

iwork

Workspace array of size \(5*n\)

minrgp

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.
OUTPUT Parameters

vl, vu
If range='V', the lower and upper bounds for the eigenvalues. Eigenvalues
less than or equal to vl, or greater than vu, are not returned. vl < vu.
If range='I' or range='A', ?larr2a computes bounds on the desired part
of the spectrum.

d
The n diagonal elements of the diagonal matrices D_i.
e
contains the subdiagonal elements of the unit bidiagonal matrices L_i. The
entries e[isplit[i]], 0 ≤ isplit, contain the base points σ_i+1 on output.
e2
The entries e2[isplit[i]], 0 ≤ isplit have been set to zero.

nsplit
The number of blocks T splits into. 1 ≤ nsplit ≤ n.
isplit
Array of size n
The splitting points, at which T breaks up into blocks.
The first block consists of rows/columns 1 to isplit[0], the second of
rows/columns isplit[0]+1 through isplit[1], etc., and the nsplit-th
block consists of rows/columns isplit[nsplit-2]+1 through
isplit[nsplit-1]=n.
m
The total number of eigenvalues (of all L_iD_iL^T_i) found.
needil, neediu
The indices of the leftmost and rightmost eigenvalues of the root node RRR
which are needed to accurately compute the relevant part of the
representation tree.
w
Array of size n
The first m elements contain the eigenvalues. The eigenvalues of each of the
blocks, L_iD_iL^T_i, are sorted in ascending order (?larr2a may use the
remaining n-m elements as workspace).
Note that immediately after exiting this function, only the eigenvalues in
w with indices in range dol-1:dou-1 rely on this processor because the
eigenvalue computation is done in parallel.
werr
Array of size n
The error bound on the corresponding eigenvalue in w.
Note that immediately after exiting this function, only the uncertainties in
werr with indices in range dol-1:dou-1 are reliable on this processor because the
eigenvalue computation is done in parallel.
wgap
Array of size n
The separation from the right neighbor eigenvalue in w. The gap is only with
respect to the eigenvalues of the same block as each block has its own
representation tree.
Exception: at the right end of a block we store the left gap
Note that immediately after exiting this function, only the gaps in wgap with
indices in range dol-1:dou-1 are reliable on this processor because the
eigenvalue computation is done in parallel.
iblock

Array of size \( n \)

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in \( w \); \( iblock[i]=1 \) if eigenvalue \( w[i] \) belongs to the first block from the top, \( iblock[i]=2 \) if \( w[i] \) belongs to the second block, and so on.

indexw

Array of size \( n \)

The indices of the eigenvalues within each block (submatrix); for example, \( indexw[i] = 10 \) and \( iblock[i]=2 \) imply that the \((i+1)\)-th eigenvalue \( w[i] \) is the 10th eigenvalue in block 2.

gers

Array of size \( 2*n \)

The \( n \) Gerschgorin intervals (the \( i \)-th Gerschgorin interval is \((gers[2*i-2], gers[2*i-1])\)).

pivmin

The minimum pivot in the sturm sequence for \( T \).

info

= 0: successful exit
> 0: A problem occurred in ?larre2a.
< 0: One of the called functions signaled an internal problem. Needs inspection of the corresponding parameter info for further information.
= -2: Not enough internal iterations to find base representation.
= -3: Problem in ?larrb2 when computing the refined root representation.
= -4: Problem in ?larrb2 when preforming bisection on the desired part of the spectrum.
= -9 Problem: \( m < \text{dou-dol+1} \), that is the code found fewer eigenvalues than it was supposed to.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

?larrf2

Finds a new relatively robust representation such that at least one of the eigenvalues is relatively isolated.

Syntax

void slarrf2(MKL_INT* n, float* d, float* l, float* ld, MKL_INT* clstrt, MKL_INT* clend, MKL_INT* clmid1, MKL_INT* clmid2, float* w, float* wgap, float* werr, MKL_INT* trymid, float* spdiam, float* clgapl, float* clgapr, float* pivmin, float* sigma, float* dplus, float* lplus, float* work, MKL_INT* info);

void dlarrf2(MKL_INT* n, double* d, double* l, double* ld, MKL_INT* clstrt, MKL_INT* clend, MKL_INT* clmid1, MKL_INT* clmid2, double* w, double* wgap, double* werr, MKL_INT* trymid, double* spdiam, double* clgapl, double* clgapr, double* pivmin, double* sigma, double* dplus, double* lplus, double* work, MKL_INT* info);

Include Files

• mkl_scalapack.h
Description

Given the initial representation $LDL^T$ and its cluster of close eigenvalues (in a relative measure), defined by the indices of the first and last eigenvalues in the cluster, ?larrf2 finds a new relatively robust representation $LDL^T - \sigma I = L\_4D\_4L\_4^T$ such that at least one of the eigenvalues of $L\_4D\_4L\_4^T$ is relatively isolated.

This is an enhanced version of ?larrf that also tries shifts in the middle of the cluster, should there be a large gap, in order to break large clusters into at least two pieces.

Input Parameters

- $n$ The order of the matrix (subblock, if the matrix was split).
- $d$ Array of size $n$
  The $n$ diagonal elements of the diagonal matrix $D$.
- $l$ Array of size $n-1$
  The ($n-1$) subdiagonal elements of the unit bidiagonal matrix $L$.
- $ld$ Array of size $n-1$
  The ($n-1$) elements $l[i] * d[i]$.
- $clstrt$ The index of the first eigenvalue in the cluster.
- $clend$ The index of the last eigenvalue in the cluster.
- $clmid1, clmid2$ The index of a middle eigenvalue pair with large gap.
- $w$ Array of size $\geq (clend - clstrt + 1)$
  The eigenvalue approximations of $LDL^T$ in ascending order. $w[clstrt - 1]$ through $w[clend - 1]$ form the cluster of relatively close eigenvalues.
- $wgap$ Array of size $\geq (clend - clstrt + 1)$
  The separation from the right neighbor eigenvalue in $w$.
- $werr$ Array of size $\geq (clend - clstrt + 1)$
  $werr$ contains the semiwidth of the uncertainty interval of the corresponding eigenvalue approximation in $w$.
- $spdiam$ Estimate of the spectral diameter obtained from the Gerschgorin intervals
- $clgapl, clgapr$ Absolute gap on each end of the cluster.
  Set by the calling function to protect against shifts too close to eigenvalues outside the cluster.
- $pivmin$ The minimum pivot allowed in the Sturm sequence.
- $work$ Workspace array of size $2*n$

Output Parameters

- $wgap$ Contains refined values of its input approximations. Very small gaps are unchanged.
- $sigma$ The shift ($\sigma$) used to form $L\_4D\_4L\_4^T$. 
**dplus**

Array of size \( n \)

The \( n \) diagonal elements of the diagonal matrix \( D_+ \).

**lplus**

Array of size \( n-1 \)

The first \((n-1)\) elements of \( lplus \) contain the subdiagonal elements of the unit bidiagonal matrix \( L_+ \).

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

`?larrv2`

**Computes the eigenvectors of the tridiagonal matrix**

\[
T = LDL^T \text{ given } L, D \text{ and the eigenvalues of } L^*D^*L^T.
\]

**Syntax**

```c
void slarrv2(MKL_INT* n, float* vl, float* vu, float* d, float* l, float* pivmin,
MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil,
MKL_INT* neediu, float* minrgp, float* rtoi1, float* rtoi2, float* w,
float* werr, float* wgap, MKL_INT* iblock, MKL_INT* indexw, float* gers,
float* sdiamp, float* z, MKL_INT* ldz, MKL_INT* isuppz, float* work,
MKL_INT* iwork, MKL_INT* vstart, MKL_INT* finish, MKL_INT* maxcls,
MKL_INT* parity, MKL_INT* zoffset, MKL_INT* info);

void dlarrv2(MKL_INT* n, double* vl, double* vu, double* d, double* l,
double* pivmin, MKL_INT* isplit, MKL_INT* m, MKL_INT* dol, MKL_INT* dou,
MKL_INT* needil, MKL_INT* neediu, double* minrgp, double* rtoi1,
double* rtoi2, double* w, double* werr, double* wgap, MKL_INT* iblock,
MKL_INT* indexw, double* gers, double* sdiamp, double* z,
MKL_INT* ldz, MKL_INT* isuppz, double* work, MKL_INT* iwork,
MKL_INT* vstart, MKL_INT* finish, MKL_INT* maxcls, MKL_INT* parity,
MKL_INT* zoffset, MKL_INT* info);
```

Include Files

- mkl_scalapack.h

**Description**

`?larrv2` computes the eigenvectors of the tridiagonal matrix \( T = LDL^T \) given \( L, D \) and approximations to the eigenvalues of \( LDL^T \). The input eigenvalues should have been computed by `larre2a` or by previous calls to `?larrv2`.

The major difference between the parallel and the sequential construction of the representation tree is that in the parallel case, not all eigenvalues of a given cluster might be computed locally. Other processors might "own" and refine part of an eigenvalue cluster. This is crucial for scalability. Thus there might be communication necessary before the current level of the representation tree can be parsed.

Please note:

- The calling sequence has two additional integer parameters, \( dol \) and \( dou \), that should satisfy \( m \geq dou \geq dol \geq 1 \). These parameters are only relevant when both eigenvalues and eigenvectors are computed (stegr2b parameter \( jobz = 'V' \)). `?larrv2` only computes the eigenvectors corresponding to eigenvalues \( dol \) through \( dou \) in \( w \). (That is, instead of computing the eigenvectors belonging to \( w[0] \) through \( w[m-1] \), only the eigenvectors belonging to eigenvalues \( w[dol - 1] \) through \( w[dou -1] \) are computed. In this case, only the eigenvalues \( dol:dou \) are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.
The additional arguments \texttt{vstart}, \texttt{finish}, \texttt{ndepth}, \texttt{parity}, \texttt{zoffset} are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable \texttt{finish} is non-zero, the computation has ended. All eigenpairs between \texttt{dol} and \texttt{dou} have been computed. \texttt{m} is set to \texttt{dou - dol + 1}.

\texttt{?larrv2} needs more workspace in \texttt{z} than the sequential \texttt{slarrv}. It is used to store the conformal embedding of the local representation tree.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

### Input Parameters

- \texttt{n} 
  The order of the matrix. \( n \geq 0 \).

- \texttt{vl, vu} 
  Lower and upper bounds of the interval that contains the desired eigenvalues. \( vl < vu \). Needed to compute gaps on the left or right end of the extremal eigenvalues in the desired range. \( vu \) is currently not used but kept as parameter in case needed.

- \texttt{d} 
  Array of size \( n \)
  The \( n \) diagonal elements of the diagonal matrix \( d \). On exit, \( d \) is overwritten.

- \texttt{l} 
  Array of size \( n \)
  The \((n-1)\) subdiagonal elements of the unit bidiagonal matrix \( L \) are in elements 0 to \( n-2 \) of \( i \) (if the matrix is not split.) At the end of each block is stored the corresponding shift as given by \texttt{?larre}. On exit, \( l \) is overwritten.

- \texttt{pivmin} 
  The minimum pivot allowed in the sturm sequence.

- \texttt{isplit} 
  Array of size \( n \)
  The splitting points, at which the matrix \( T \) breaks up into blocks. The first block consists of rows/columns 1 to \texttt{isplit[0]}, the second of rows/columns \texttt{isplit[0]} + 1 through \texttt{isplit[1]}, etc.

- \texttt{m} 
  The total number of input eigenvalues. \( 0 \leq m \leq n \).

- \texttt{dol, dou} 
  If you want to compute only selected eigenvectors from all the eigenvalues supplied, you can specify an index range \texttt{dol:dou}. Or else the setting \texttt{dol=1, dou=m} should be applied. Note that \texttt{dol} and \texttt{dou} refer to the order in which the eigenvalues are stored in \texttt{w}. If you want to compute only
selected eigenpairs, the columns $dol-1$ to $dou+1$ of the eigenvector space $Z$ contain the computed eigenvectors. All other columns of $Z$ are set to zero.

If $dol > 1$, then $Z(:,dol-1-zoffset)$ is used.

If $dou < n$, then $Z(:,dou+1-zoffset)$ is used.

$needil, neediu$

Describe which are the left and right outermost eigenvalues that still need to be included in the computation. These indices indicate whether eigenvalues from other processors are needed to correctly compute the conformally embedded representation tree.

When $dol\leq needil\leq neediu\leq dou$, all required eigenvalues are local to the processor and no communication is required to compute its part of the representation tree.

$minrgp$

The minimum relative gap threshold to decide whether an eigenvalue or a cluster boundary is reached.

$rtol1, rtol2$

Parameters for bisection. An interval $[left, right]$ has converged if $right-left < \max( rtol1*gap, rtol2*\max(|left|,|right|) )$.

$w$

Array of size $n$

The first $m$ elements of $w$ contain the approximate eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array $w$ from $?stegr2a$ is expected here.) Furthermore, they are with respect to the shift of the corresponding root representation for their block.

$werr$

Array of size $n$

The first $m$ elements contain the semiwidth of the uncertainty interval of the corresponding eigenvalue in $w$.

$wgap$

Array of size $n$

The separation from the right neighbor eigenvalue in $w$.

$iblock$

Array of size $n$

The indices of the blocks (submatrices) associated with the corresponding eigenvalues in $w$; $iblock[i]=1$ if eigenvalue $w[i]$ belongs to the first block from the top, $iblock[i]=2$ if $w[i]$ belongs to the second block, and so on.

$indexw$

Array of size $n$

The indices of the eigenvalues within each block (submatrix). For example: $indexw[i]= 10$ and $iblock[i]=2$ imply that the $(i+1)$-th eigenvalue $w[i]$ is the $10$th eigenvalue in block $2$.

$gers$

Array of size $2*n$

The $n$ Gerschgorin intervals (the $i$-th Gerschgorin interval is $(gers[2*i-2], gers[2*i-1])$). The Gerschgorin intervals should be computed from the original unshifted matrix.

Not used but kept as parameter for possible future use.
sdiam

Array of size \( n \)
The spectral diameters for all unreduced blocks.

ldz

The leading dimension of the array \( z \). \( ldz \geq 1 \), and if stegr2b parameter jobz = 'V', \( ldz \geq \max(1,n) \).

work

(workspace) array of size \( 12*n \)

iwork

(workspace) Array of size \( 7*n \)

vstart

Non-zero on initialization, set to zero afterwards.

finish

A flag that indicates whether all eigenpairs have been computed.

maxcls

The largest cluster worked on by this processor in the representation tree.

ndepth

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

parity

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

zoffset

Offset for storing the eigenpairs when \( z \) is distributed in 1D-cyclic fashion.

OUTPUT Parameters

needil, neediu

Unshifted eigenvalues for which eigenvectors have already been computed.

w

Contains refined values of its input approximations.

werr

Contains refined values of its input approximations. Very small gaps are changed.

wgap

Array of size \( ldz \times \max(1,m) \)

If info = 0, the first \( m \) columns of the matrix \( Z \), stored in the array \( z \), contain the orthonormal eigenvectors of the matrix \( T \) corresponding to the input eigenvalues, with the \( i \)-th column of \( Z \) holding the eigenvector associated with \( \lambda[i-1] \).

In the distributed version, only a subset of columns is accessed, see dol, dou, and zoffset.

isuppz

Array of size \( 2 \times \max(1,m) \)
The support of the eigenvectors in \( z \), i.e., the indices indicating the non-zero elements in \( z \). The \( i \)-th eigenvector is non-zero only in elements \( isuppz[2*i-2] \) through \( isuppz[2*i-1] \).

vstart

Non-zero on initialization, set to zero afterwards.

finish

A flag that indicates whether all eigenpairs have been computed.

maxcls

The largest cluster worked on by this processor in the representation tree.

ndepth

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.
**parity**

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

**info**

- 0: successful exit
- > 0: A problem occurred in ?larrv2.
- < 0: One of the called functions signaled an internal problem. Needs inspection of the corresponding parameter info for further information.
- -1: Problem in ?larrb2 when refining a child's eigenvalues.
- -2: Problem in ?larrf2 when computing the RRR of a child. When a child is inside a tight cluster, it can be difficult to find an RRR. A partial remedy from the user's point of view is to make the parameter minrgp smaller and recompile. However, as the orthogonality of the computed vectors is proportional to 1/minrgp, be aware that decreasing minrgp might be a reduction in precision.
- -3: Problem in ?larrb2 when refining a single eigenvalue after the Rayleigh correction was rejected.
- 5: The Rayleigh Quotient Iteration failed to converge to full accuracy.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?lasorte

*Sorts eigenpairs by real and complex data types.*

**Syntax**

```c
void slasorte (float *s, MKL_INT *lds, MKL_INT *j, float *out, MKL_INT *info);
void dlasorte (double *s, MKL_INT *lds, MKL_INT *j, double *out, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The ?lasorte function sorts eigenpairs so that real eigenpairs are together and complex eigenpairs are together. This helps to employ 2x2 shifts easily since every second subdiagonal is guaranteed to be zero. This function does no parallel work and makes no calls.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
**Input Parameters**

- **s** (local)
  - Array of size lds.
  - On entry, a matrix already in Schur form.

- **lds** (local)
  - On entry, the leading dimension of the array s; unchanged on exit.

- **j** (local)
  - On entry, the order of the matrix S; unchanged on exit.

- **out** (local)
  - Array of size 2*j. The work buffer required by the function.

- **info** (local)
  - Set, if the input matrix had an odd number of real eigenvalues and things could not be paired or if the input matrix S was not originally in Schur form. 0 indicates successful completion.

**Output Parameters**

- **s**
  - On exit, the diagonal blocks of S have been rewritten to pair the eigenvalues. The resulting matrix is no longer similar to the input.

- **out**
  - Work buffer.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

### ?lasrt2

**Sorts numbers in increasing or decreasing order.**

**Syntax**

```c
void slasrt2 (char *id, MKL_INT *n, float *d, MKL_INT *key, MKL_INT *info);
void dlasrt2 (char *id, MKL_INT *n, double *d, MKL_INT *key, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The ?lasrt2 function is modified LAPACK function ?lasrt, which sorts the numbers in d in increasing order (if id = 'I') or in decreasing order (if id = 'D'). It uses Quick Sort, reverting to Insertion Sort on arrays of size ≤ 20. The size of STACK limits n to about $2^{32}$.

**Input Parameters**

- **id**
  - = 'I': sort d in increasing order;
  - = 'D': sort d in decreasing order.
The length of the array \( d \).

\( d \)

Array of size \( n \).

On entry, the array to be sorted.

\( key \)

Array of size \( n \).

On entry, \( key \) contains a key to each of the entries in \( d \).

Typically, \( key[i] = i+1 \) for all \( i = 0, ..., n-1 \).

Output Parameters

\( d \)

On exit, \( d \) has been sorted into increasing order

\( (d[0] \leq ... \leq d[n-1]) \)

or into decreasing order

\( (d[0] \geq ... \geq d[n-1]) \),

depending on \( id \).

\( info \)

\( = 0 \): successful exit

\(< 0 \): if \( info = -i \), the \( i \)-th argument had an illegal value.

\( key \)

On exit, \( key \) is permuted in exactly the same manner as \( d \) was permuted from input to output. Therefore, if \( key[i] = i+1 \) for all \( i = 0, ..., n-1 \) on input, \( d[i] \) on output equals \( d[\text{key}[i]-1] \) on input.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

\(?\text{stegr2}\)

Computes selected eigenvalues and eigenvectors of a real symmetric tridiagonal matrix.

Syntax

\[
\text{void sstegr2}(\text{char* jobz, char* range, MKL_INT* n, float* d, float* e, float* vl, float* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, float* w, float* z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* zoffset, MKL_INT* info});
\]

\[
\text{void dstegr2}(\text{char* jobz, char* range, MKL_INT* n, double* d, double* e, double* vl, double* vu, MKL_INT* il, MKL_INT* iu, MKL_INT* m, double* w, double* z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* zoffset, MKL_INT* info});
\]

Include Files

- \text{mkl_scalapack.h}

Description

\(?\text{stegr2}\) computes selected eigenvalues and, optionally, eigenvectors of a real symmetric tridiagonal matrix \( T \). It is invoked in the ScaLAPACK MRRR driver \( p\text{syevr} \) and the corresponding Hermitian version either when only eigenvalues are to be computed, or when only a single processor is used (the sequential-like case).

\(?\text{stegr2}\) has been adapted from LAPACK's \( \text{stegr} \). Please note the following crucial changes.
1. The calling sequence has two additional integer parameters, \( \text{dol} \) and \( \text{dou} \), that should satisfy \( \text{m} \geq \text{dou} \geq \text{dol} \geq 1 \). ?steqr2 only computes the eigenpairs corresponding to eigenvalues \( \text{dol} \) through \( \text{dou} \) in \( w \), indexed \( \text{dol}-1 \) through \( \text{dou}-1 \). (That is, instead of computing the eigenpairs belonging to \( w[0] \) through \( w[m-1] \), only the eigenvectors belonging to eigenvalues \( w[\text{dol}-1] \) through \( w[\text{dou}-1] \) are computed. In this case, only the eigenvalues \( \text{dol} \) through \( \text{dou} \) are guaranteed to be fully accurate.

2. \( m \) is not the number of eigenvalues specified by \( \text{range} \), but is \( m = \text{dou} - \text{dol} + 1 \). This concerns the case where only eigenvalues are computed, but on more than one processor. Thus, in this case \( m \) refers to the number of eigenvalues computed on this processor.

3. The arrays \( w \) and \( z \) might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

- **jobz**
  - = 'N': Compute eigenvalues only;
  - = 'V': Compute eigenvalues and eigenvectors.

- **range**
  - = 'A': all eigenvalues will be found.
  - = 'V': all eigenvalues in the half-open interval \((vl,vu]\) will be found.
  - = 'I': eigenvalues of the entire matrix with the indices in a given range will be found.

- **n**
  The order of the matrix. \( n \geq 0 \).

- **d**
  Array of size \( n \)
  On entry, the \( n \) diagonal elements of the tridiagonal matrix \( T \). Overwritten on exit.

- **e**
  Array of size \( n \)
  On entry, the \((n-1)\) subdiagonal elements of the tridiagonal matrix \( T \) in elements 0 to \( n-2 \) of \( e[\text{e}-1] \) need not be set on input, but is used internally as workspace. Overwritten on exit.

- **vl**
  - If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues. \( vl < vu \).
  - Not referenced if \( \text{range} = 'A' \) or \( 'I' \).

- **vu**
  - If \( \text{range} = 'V' \), the lower and upper bounds of the interval to be searched for eigenvalues. \( vl < vu \).
  - Not referenced if \( \text{range} = 'A' \) or \( 'I' \).

- **il, iu**
  - If \( \text{range} = 'I' \), the indices (in ascending order) of the smallest eigenvalue, to be returned in \( w[\text{il}-1] \), and largest eigenvalue, to be returned in \( w[\text{iu}-1] \).
$1 \leq i_{ll} \leq i_{ul} \leq n$, if $n > 0$.

Not referenced if range = 'A' or 'V'.

ldz

The leading dimension of the array $z$. $ldz \geq 1$, and if jobz = 'V', then $ldz \geq \text{max}(1,n)$.

nzc

The number of eigenvectors to be held in the array $z$, storing the matrix $Z$.

If range = 'A', then $nzc \geq \text{max}(1,n)$.

If range = 'V', then $nzc \geq$ the number of eigenvalues in $(vl, vu]$.

If range = 'I', then $nzc \geq i_{ul} - i_{ll} + 1$.

If $nzc = -1$, then a workspace query is assumed; the function calculates the number of columns of the matrix $Z$ that are needed to hold the eigenvectors. This value is returned as the first entry of the $z$ array, and no error message related to $nzc$ is issued.

lwork

The size of the array work. $lwork \geq \text{max}(1, 18*n)$ if jobz = 'V', and $lwork \geq \text{max}(1, 12*n)$ if jobz = 'N'. If $lwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the work array, returns this value as the first entry of the work array, and no error message related to $lwork$ is issued.

liwork

The size of the array liwork. $liwork \geq \text{max}(1, 10*n)$ if the eigenvectors are desired, and $liwork \geq \text{max}(1, 8*n)$ if only the eigenvalues are to be computed.

If $liwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the liwork array, returns this value as the first entry of the liwork array, and no error message related to liwork is issued.

dol, dou

From the eigenvalues $w[0]$ through $w[n-1]$, only eigenvectors $Z(:,dol)$ to $Z(:,dou)$ are computed.

If $dol > 1$, then $Z(:,dol-1-zoffset)$ is used and overwritten.

If $dou < m$, then $Z(:,dou+1-zoffset)$ is used and overwritten.

zoffset

Offset for storing the eigenpairs when $z$ is distributed in 1D-cyclic fashion

OUTPUT Parameters

m

Globally summed over all processors, $m$ equals the total number of eigenvalues found. $0 \leq m \leq n$. If range = 'A', $m = n$, and if range = 'I', $m = i_{ul} - i_{ll} + 1$. The local output equals $m = dou - dol + 1$.

w

Array of size $n$

The first $m$ elements contain the selected eigenvalues in ascending order. Note that immediately after exiting this function, only the eigenvalues indexed $dol-1$ through $dou-1$ are reliable on this processor because the eigenvalue computation is done in parallel. Other processors will hold reliable information on other parts of the $w$ array. This information is communicated in the ScaLAPACK driver.

z

Array of size $ldz \times \text{max}(1,m)$. 

1714
If $jobz = 'V'$, and if $info = 0$, then the first $m$ columns of the matrix $Z$ stored in $z$ contain some of the orthonormal eigenvectors of the matrix $T$ corresponding to the selected eigenvalues, with the $i$-th column of $Z$ holding the eigenvector associated with $w[i-1]$.

If $jobz = 'N'$, then $z$ is not referenced.

Note: the user must ensure that at least $\max(1, m)$ columns of the matrix are supplied in the array $z$; if $range = 'V'$, the exact value of $m$ is not known in advance and can be computed with a workspace query by setting $nzc = -1$, see below.

$\text{isuppz}$ array of size $2*\max(1, m)$

The support of the eigenvectors in $z$, i.e., the indices indicating the nonzero elements in $z$. The $i$-th computed eigenvector is nonzero only in elements $\text{isuppz}[2*i-2]$ through $\text{isuppz}[2*i-1]$. This is relevant in the case when the matrix is split. $\text{isuppz}$ is only set if $n>2$.

$\text{work}$

On exit, if $info = 0$, $\text{work}[0]$ returns the optimal (and minimal) $lwork$.

$\text{iwork}$

On exit, if $info = 0$, $\text{iwork}[0]$ returns the optimal $liwork$.

$\text{info}$

On exit, $info$

= 0: successful exit
other: if $info = -i$, the $i$-th argument had an illegal value
if $info = 10X$, internal error in $\text{?larre2}$,
if $info = 20X$, internal error in $\text{?larrv}$.

Here, the digit $X = \text{ABS}(iinfo) < 10$, where $iinfo$ is the nonzero error code returned by $\text{?larre2}$ or $\text{?larrv}$, respectively.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

$\text{?stegr2a}$

Computes selected eigenvalues and initial representations needed for eigenvector computations.

Syntax

void $\text{sstegr2a}(\text{char* } jobz, \text{char* } range, \text{MKL_INT* } n, \text{float* } d, \text{float* } e, \text{float* } vl, \text{float* } vu, \text{MKL_INT* } il, \text{MKL_INT* } iu, \text{MKL_INT* } m, \text{float* } w, \text{float* } z, \text{MKL_INT* } ldz, \text{MKL_INT* } nzc, \text{float* } work, \text{MKL_INT* } lwork, \text{MKL_INT* } iwork, \text{MKL_INT* } liwork, \text{MKL_INT* } dol, \text{MKL_INT* } dou, \text{MKL_INT* } needil, \text{MKL_INT* } neediu, \text{MKL_INT* } inderr, \text{MKL_INT* } nsplit, \text{float* } pivmin, \text{float* } scale, \text{float* } wl, \text{float* } wu, \text{MKL_INT* } info);$ 

void $\text{dstegr2a}(\text{char* } jobz, \text{char* } range, \text{MKL_INT* } n, \text{double* } d, \text{double* } e, \text{double* } vl, \text{double* } vu, \text{MKL_INT* } il, \text{MKL_INT* } iu, \text{MKL_INT* } m, \text{double* } w, \text{double* } z, \text{MKL_INT* } ldz, \text{MKL_INT* } nzc, \text{double* } work, \text{MKL_INT* } lwork, \text{MKL_INT* } iwork, \text{MKL_INT* } liwork, \text{MKL_INT* } dol, \text{MKL_INT* } dou, \text{MKL_INT* } needil, \text{MKL_INT* } neediu, \text{MKL_INT* } inderr, \text{MKL_INT* } nsplit, \text{double* } pivmin, \text{double* } scale, \text{double* } wl, \text{double* } wu, \text{MKL_INT* } info);$
Include Files

- mkl_scalapack.h

Description

?stegr2a computes selected eigenvalues and initial representations needed for eigenvector computations in ?stegr2b. It is invoked in the ScaLAPACK MRRR driver ?syevr and the corresponding Hermitian version when both eigenvalues and eigenvectors are computed in parallel on multiple processors. For this case, ?stegr2a implements the first part of the MRRR algorithm, parallel eigenvalue computation and finding the root RRR. At the end of ?stegr2a, other processors might have a part of the spectrum that is needed to continue the computation locally. Once this eigenvalue information has been received by the processor, the computation can then proceed by calling the second part of the parallel MRRR algorithm, ?stegr2b.

Please note:

- The calling sequence has two additional integer parameters, (compared to LAPACK’s ?stegr), these are dol and dou and should satisfy \( m \geq dou \geq dol \geq 1 \). These parameters are only relevant for the case jobz = 'V'.

Globally invoked over all processors, ?stegr2a computes all the eigenvalues specified by range.

?stegr2a locally only computes the eigenvalues corresponding to eigenvalues dol through dou in \( \omega \), indexed \( dol-1 \) through \( dou-1 \). (That is, instead of computing the eigenvectors belonging to \( \omega[m-1] \), only the eigenvectors belonging to eigenvalues \( \omega[dol-1] \) through \( \omega[dou-1] \) are computed. In this case, only the eigenvalues dol through dou are guaranteed to be fully accurate.

- \( m \) is not the number of eigenvalues specified by range, but it is \( m = dou - dol + 1 \). Instead, \( m \) refers to the number of eigenvalues computed on this processor.

- While no eigenvectors are computed in ?stegr2a itself (this is done later in ?stegr2b), the interface

  If jobz = 'V' then, depending on range and dol, dou, ?stegr2a might need more workspace in z then the original ?stegr. In particular, the arrays \( \omega \) and \( z \) might not contain all the wanted eigenpairs locally, instead this information is distributed over other processors.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

- **jobz**
  
  = 'N': Compute eigenvalues only;
  
  = 'V': Compute eigenvalues and eigenvectors.

- **range**
  
  = 'A': all eigenvalues will be found.
  
  = 'V': all eigenvalues in the half-open interval \((vl, vu]\) will be found.
  
  = 'I': eigenvalues of the entire matrix with the indices in a given range will be found.

- **n**
  
  The order of the matrix. \( n \geq 0 \).
**d**

Array of size $n$

The $n$ diagonal elements of the tridiagonal matrix $T$. Overwritten on exit.

**e**

Array of size $n$

On entry, the $(n-1)$ subdiagonal elements of the tridiagonal matrix $T$ in elements 0 to $n-2$ of $e$. $e[n-1]$ need not be set on input, but is used internally as workspace. Overwritten on exit.

**vl, vu**

If $range = 'V'$, the lower and upper bounds of the interval to be searched for eigenvalues. $vl < vu$.

Not referenced if $range = 'A'$ or 'I'.

**il, iu**

If $range = 'I'$, the indices (in ascending order) of the smallest eigenvalue, to be returned in $w[il-1]$, and largest eigenvalue, to be returned in $w[iu-1]$. $1 \leq il \leq iu \leq n$, if $n > 0$.

Not referenced if $range = 'A'$ or 'V'.

**ldz**

The leading dimension of the array $z$. $ldz \geq 1$, and if $jobz = 'V'$, then $ldz \geq \max(1,n)$.

**nzc**

The number of eigenvectors to be held in the array $z$.

If $range = 'A'$, then $nzc \geq \max(1,n)$.

If $range = 'V'$, then $nzc \geq$ the number of eigenvalues in $(vl,vu)$.

If $range = 'I'$, then $nzc \geq iu-il+1$.

If $nzc = -1$, then a workspace query is assumed; the function calculates the number of columns of the matrix stored in array $z$ that are needed to hold the eigenvectors. This value is returned as the first entry of the $z$ array, and no error message related to $nzc$ is issued.

**lwork**

The size of the array $work$. $lwork \geq \max(1,18*n)$ if $jobz = 'V'$, and $lwork \geq \max(1,12*n)$ if $jobz = 'N'$.

If $lwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $work$ array, returns this value as the first entry of the $work$ array, and no error message related to $lwork$ is issued.

**liwork**

The size of the array $iwork$. $liwork \geq \max(1,10*n)$ if the eigenvectors are desired, and $liwork \geq \max(1,8*n)$ if only the eigenvalues are to be computed.

If $liwork = -1$, then a workspace query is assumed; the function only calculates the optimal size of the $iwork$ array, returns this value as the first entry of the $iwork$ array, and no error message related to $liwork$ is issued.

**dol, dou**

From all the eigenvalues $w[0]$ through $w[m-1]$, only eigenvalues $w[dol-1]$ through $w[dou-1]$ are computed.

**OUTPUT Parameters**

**m**

Globally summed over all processors, $m$ equals the total number of eigenvalues found. $0 \leq m \leq n$. 
If \( \text{range} = 'A' \), \( m = n \), and if \( \text{range} = 'I' \), \( m = iu-il+1 \).

The local output equals \( m = dou-dol+1 \).

\( w \)

Array of size \( n \)

The first \( m \) elements contain approximations to the selected eigenvalues in ascending order. Note that immediately after exiting this function, only the eigenvalues indexed \( dol-1 \) through \( dou-1 \) are reliable on this processor because the eigenvalue computation is done in parallel. The other entries are very crude preliminary approximations. Other processors hold reliable information on these other parts of the \( w \) array.

This information is communicated in the ScaLAPACK driver.

\( z \)

Array of size \( ldz \times \max(1,m) \).

\texttt{?stegr2a} does not compute eigenvectors, this is done in \texttt{?stegr2b}. The argument \( z \) as well as all related other arguments only appear to keep the interface consistent and to signal to the user that this function is meant to be used when eigenvectors are computed.

\( \text{work} \)

On exit, if \( \text{info} = 0 \), \( \text{work}[0] \) returns the optimal (and minimal) \( lwork \).

\( \text{iwork} \)

On exit, if \( \text{info} = 0 \), \( \text{iwork}[0] \) returns the optimal \( liwork \).

\( \text{needil, neediu} \)

The indices of the leftmost and rightmost eigenvalues needed to accurately compute the relevant part of the representation tree. This information can be used to find out which processors have the relevant eigenvalue information needed so that it can be communicated.

\( \text{inderr} \)

\( \text{inderr} \) points to the place in the work space where the eigenvalue uncertainties (errors) are stored.

\( \text{nsplit} \)

The number of blocks into which \( T \) splits. \( 1 \leq \text{nsplit} \leq n \).

\( \text{pivmin} \)

The minimum pivot in the sturm sequence for \( T \).

\( \text{scale} \)

The scaling factor for the tridiagonal \( T \).

\( \text{wl, wu} \)

The interval \((\text{wl}, \text{wu}]\) contains all the wanted eigenvalues.

It is either given by the user or computed in \texttt{?larre2a}.

\( \text{info} \)

On exit, \( \text{info} = 0 \): successful exit

other: if \( \text{info} = -i \), the \( i \)-th argument had an illegal value

if \( \text{info} = 10x \), internal error in \texttt{?larre2a},

Here, the digit \( x = \text{abs} \left( \text{iinfo} \right) < 10 \), where \( \text{iinfo} \) is the nonzero error code returned by \texttt{?larre2a}.

\textbf{See Also}

\textbf{Overview} for details of ScaLAPACK array descriptor structures and related notations.
**ssteqr2b**

From eigenvalues and initial representations computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors.

**Syntax**

```c
void sstegr2b(char* jobz, MKL_INT* n, float* d, float* e, MKL_INT* m, float* w, float* z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, float* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, MKL_INT* indwlc, float* pivmin, float* scale, float* wI, float* wu, MKL_INT* vstart, MKL_INT* finish, MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT* zoffset, MKL_INT* info);

void dstegr2b(char* jobz, MKL_INT* n, double* d, double* e, MKL_INT* m, double* w, double* z, MKL_INT* ldz, MKL_INT* nzc, MKL_INT* isuppz, double* work, MKL_INT* lwork, MKL_INT* iwork, MKL_INT* liwork, MKL_INT* dol, MKL_INT* dou, MKL_INT* needil, MKL_INT* neediu, MKL_INT* indwlc, double* pivmin, double* scale, double* wI, double* wu, MKL_INT* vstart, MKL_INT* finish, MKL_INT* maxcls, MKL_INT* ndepth, MKL_INT* parity, MKL_INT* zoffset, MKL_INT* info);
```

**Include Files**

- mkl_scalapack.h

**Description**

`ssteqr2b` should only be called after a call to `ssteqr2a`. From eigenvalues and initial representations computed by `ssteqr2a`, `ssteqr2b` computes the selected eigenvalues and eigenvectors of the real symmetric tridiagonal matrix in parallel on multiple processors. It is potentially invoked multiple times on a given processor because the locally relevant representation tree might depend on spectral information that is "owned" by other processors and might need to be communicated.

Please note:

- The calling sequence has two additional integer parameters, `dol` and `dou`, that should satisfy \( m \geq dou > dol > 1 \). These parameters are only relevant for the case `jobz = 'V'`. `ssteqr2b` only computes the eigenvectors corresponding to eigenvalues `dol` through `dou` in `w`, indexed `dol-1` through `dou-1`. (That is, instead of computing the eigenvectors belonging to `w[0]` through `w[m-1]`, only the eigenvectors belonging to eigenvalues `w[dol-1]` through `w[dou-1]` are computed. In this case, only the eigenvalues `dol` through `dou` are guaranteed to be accurately refined to all figures by Rayleigh-Quotient iteration.

- The additional arguments `vstart`, `finish`, `ndepth`, `parity`, `zoffset` are included as a thread-safe implementation equivalent to save variables. These variables store details about the local representation tree which is computed layerwise. For scalability reasons, eigenvalues belonging to the locally relevant representation tree might be computed on other processors. These need to be communicated before the inspection of the RRRs can proceed on any given layer. Note that only when the variable `finish` is non-zero, the computation has ended. All eigenpairs between `dol` and `dou` have been computed. `m` is set to `dou - dol + 1`.

- `ssteqr2b` needs more workspace in `z` than the sequential `ssteqr`. It is used to store the conformal embedding of the local representation tree.
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Input Parameters

jobz
- \texttt{='N'}: Compute eigenvalues only;
- \texttt{='V'}: Compute eigenvalues and eigenvectors.

n
The order of the matrix. \( n \geq 0 \).

d
Array of size \( n \)
The \( n \) diagonal elements of the tridiagonal matrix \( T \). Overwritten on exit.
e
Array of size \( n \)
The \((n-1)\) subdiagonal elements of the tridiagonal matrix \( T \) in elements 0 to \( n-2 \) of \( e \). \( e[n-1] \) need not be set on input, but is used internally as workspace. Overwritten on exit.

m
The total number of eigenvalues found in \texttt{?steqr2a}. \( 0 \leq m \leq n \).

w
Array of size \( n \)
The first \( m \) elements contain approximations to the selected eigenvalues in ascending order. Note that only the eigenvalues from the locally relevant part of the representation tree, that is all the clusters that include eigenvalues from \texttt{dol} through \texttt{dou}, are reliable on this processor. (It does not need to know about any others anyway.)

ldz
The leading dimension of the array \( z \). \( ldz \geq 1 \), and if \texttt{jobz = 'V'}, then \( ldz \geq \text{max}(1,n) \).

nzc
The number of eigenvectors to be held in the array \( z \), storing the matrix \( Z \).

lwork
The size of the array \( \text{work} \). \( lwork \geq \text{max}(1,18*n) \)
if \texttt{jobz = 'V'}, and \( lwork \geq \text{max}(1,12*n) \) if \texttt{jobz = 'N'}.

If \texttt{lwork = -1}, then a workspace query is assumed; the function only calculates the optimal size of the \texttt{work} array, returns this value as the first entry of the \texttt{work} array, and no error message related to \texttt{lwork} is issued.

liwork
The size of the array \( \text{liwork} \). \( liwork \geq \text{max}(1,10*n) \) if the eigenvectors are desired, and \( liwork \geq \text{max}(1,8*n) \) if only the eigenvalues are to be computed.
If \( liwork = -1 \), then a workspace query is assumed; the function only calculates the optimal size of the \( iwork \) array, returns this value as the first entry of the \( iwork \) array, and no error message related to \( liwork \) is issued.

\[ dol, dou \]

From the eigenvalues \( w[0] \) through \( w[m-1] \), only eigenvectors \( Z(:,dol) \) to \( Z(:,dou) \) are computed.

If \( dol > 1 \), then \( Z(:,dol-1-zoffset) \) is used and overwritten.

If \( dou < m \), then \( Z(:,dou+1-zoffset) \) is used and overwritten.

\[ needil, neediu \]

Describes which are the left and right outermost eigenvalues still to be computed. Initially computed by \(?larre2a\), modified in the course of the algorithm.

\[ pivmin \]

The minimum pivot in the sturm sequence for \( T \).

\[ scale \]

The scaling factor for \( T \). Used for unscaling the eigenvalues at the very end of the algorithm.

\[ wl, wu \]

The interval \( [wl, wu] \) contains all the wanted eigenvalues.

\[ vstart \]

Non-zero on initialization, set to zero afterwards.

\[ finish \]

Indicates whether all eigenpairs have been computed.

\[ maxcls \]

The largest cluster worked on by this processor in the representation tree.

\[ ndepth \]

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

\[ parity \]

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

\[ zoffset \]

Offset for storing the eigenpairs when \( z \) is distributed in 1D-cyclic fashion.

**OUTPUT Parameters**

\[ z \]

Array of size \( ldz * \max(1,m) \)

If \( jobz = 'V' \), and if \( info = 0 \), then a subset of the first \( m \) columns of the matrix \( Z \), stored in \( z \), contain the orthonormal eigenvectors of the matrix \( T \) corresponding to the selected eigenvalues, with the \( i \)-th column of \( Z \) holding the eigenvector associated with \( w[i-1] \).

See \( dol, dou \) for more information.

\[ isuppz \]

array of size \( 2\max(1,m) \).

The support of the eigenvectors in \( z \), i.e., the indices indicating the nonzero elements in \( z \). The \( i \)-th computed eigenvector is nonzero only in elements \( isuppz[2*i-2] \) through \( isuppz[2*i-1] \). This is relevant in the case when the matrix is split. \( isuppz \) is only set if \( n>2 \).

\[ work \]

On exit, if \( info = 0 \), \( work[0] \) returns the optimal (and minimal) \( lwork \).

\[ iwork \]

On exit, if \( info = 0 \), \( iwork[0] \) returns the optimal \( liwork \).

\[ needil, neediu \]

Modified in the course of the algorithm.
**indwlc**

Pointer into the workspace location where the local eigenvalue representations are stored. ("Local eigenvalues" are those relative to the individual shifts of the RRRs.)

**vstart**

Non-zero on initialization, set to zero afterwards.

**finish**

Indicates whether all eigenpairs have been computed.

**maxcls**

The largest cluster worked on by this processor in the representation tree.

**ndepth**

The current depth of the representation tree. Set to zero on initial pass, changed when the deeper levels of the representation tree are generated.

**parity**

An internal parameter needed for the storage of the clusters on the current level of the representation tree.

**info**

On exit, info

= 0: successful exit

other: if info = -i, the i-th argument had an illegal value

if info = 20x, internal error in ?larrv2.

Here, the digit x = abs( iinfo ) < 10, where iinfo is the nonzero error code returned by ?larrv2.

**See Also**

*Overview* for details of ScaLAPACK array descriptor structures and related notations.

---

**?stein2**

*Computes the eigenvectors corresponding to specified eigenvalues of a real symmetric tridiagonal matrix, using inverse iteration.*

**Syntax**

```c
void sstein2 (MKL_INT *n, float *d, float *e, MKL_INT *m, float *w, MKL_INT *iblock, MKL_INT *isplit, float *orfac, float *z, MKL_INT *ldz, float *work, MKL_INT *iwork, MKL_INT *ifail, MKL_INT *info );
void dstein2 (MKL_INT *n, double *d, double *e, MKL_INT *m, double *w, MKL_INT *iblock, MKL_INT *isplit, double *orfac, double *z, MKL_INT *ldz, double *work, MKL_INT *iwork, MKL_INT *ifail, MKL_INT *info );
```

**Include Files**

- mkl_scalapack.h

**Description**

The ?stein2 function is a modified LAPACK function ?stein. It computes the eigenvectors of a real symmetric tridiagonal matrix $T$ corresponding to specified eigenvalues, using inverse iteration.

The maximum number of iterations allowed for each eigenvector is specified by an internal parameter maxits (currently set to 5).

**Input Parameters**

- $n$ The order of the matrix $T$ ($n \geq 0$).
The number of eigenvectors to be found \( (0 \leq m \leq n) \).

Arrays:

\( d \), of size \( n \). The \( n \) diagonal elements of the tridiagonal matrix \( T \).

\( e \), of size \( n \).

The \( (n-1) \) subdiagonal elements of the tridiagonal matrix \( T \), in elements 1 to \( n-1 \). \( e[n-1] \) need not be set.

\( w \), of size \( n \).

The first \( m \) elements of \( w \) contain the eigenvalues for which eigenvectors are to be computed. The eigenvalues should be grouped by split-off block and ordered from smallest to largest within the block. (The output array \( w \) from \texttt{?stebz} with ORDER = 'B' is expected here).

The size of \( w \) must be at least \( \max(1, n) \).

\( i\text{block} \)

Array of size \( n \).

The submatrix indices associated with the corresponding eigenvalues in \( w \); 

\( i\text{block}[i] = 1 \), if eigenvalue \( w[i] \) belongs to the first submatrix from the top,

\( i\text{block}[i] = 2 \), if eigenvalue \( w[i] \) belongs to the second submatrix, etc. (The output array \( i\text{block} \) from \texttt{?stebz} is expected here).

\( i\text{split} \)

Array of size \( n \).

The splitting points, at which \( T \) breaks up into submatrices. The first submatrix consists of rows/columns 1 to \( i\text{split}[0] \), the second submatrix consists of rows/columns \( i\text{split}[0]+1 \) through \( i\text{split}[1] \), etc. (The output array \( i\text{split} \) from \texttt{?stebz} is expected here).

\( \text{orfac} \)

\( \text{orfac} \) specifies which eigenvectors should be orthogonalized. Eigenvectors that correspond to eigenvalues which are within \( \text{orfac} \| T \| \) of each other are to be orthogonalized.

\( \text{ldz} \)

The leading dimension of the output array \( z \); \( \text{ldz} \geq \max(1, n) \).

\( \text{work} \)

Workspace array of size \( 5n \).

\( i\text{work} \)

Workspace array of size \( n \).

**Output Parameters**

\( z \)

Arrary of size \( \text{ldz} \times m \).

The computed eigenvectors. The eigenvector associated with the eigenvalue \( w[i] \) is stored in the \((i+1)\)-th column of the matrix \( Z \) represented by \( z \), \( i=0, ..., m-1 \). Any vector that fails to converge is set to its current iterate after \( \text{maxits} \) iterations.

\( i\text{fail} \)

Arrary of size \( m \).
On normal exit, all elements of `ifail` are zero. If one or more eigenvectors fail to converge after `maxits` iterations, then their indices are stored in the array `ifail`.

\[ info = 0, \text{the exit is successful.} \]
\[ info < 0: \text{if } info = -i, \text{the } i\text{-th had an illegal value.} \]
\[ info > 0: \text{if } info = i, \text{then } i \text{ eigenvectors failed to converge in } maxits \text{ iterations. Their indices are stored in the array } ifail. \]

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**?dbtf2**

*Computes an LU factorization of a general band matrix with no pivoting (local unblocked algorithm).*

**Syntax**

```c
void sdbtf2 (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, float *ab, MKL_INT *ldab, MKL_INT *info);
void ddbtf2 (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, double *ab, MKL_INT *ldab, MKL_INT *info);
void cdbtf2 (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, MKL_Complex8 *ab, MKL_INT *ldab, MKL_INT *info);
void zdbtf2 (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, MKL_Complex16 *ab, MKL_INT *ldab, MKL_INT *info);
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `?dbtf2` function computes an LU factorization of a general real/complex \( m \)-by-\( n \) band matrix \( A \) without using partial pivoting with row interchanges.

This is the unblocked version of the algorithm, calling BLAS Routines and Functions.

**Input Parameters**

- `m`
  The number of rows of the matrix \( A (m \geq 0) \).

- `n`
  The number of columns in \( A (n \geq 0) \).

- `kl`
  The number of sub-diagonals within the band of \( A (kl \geq 0) \).

- `ku`
  The number of super-diagonals within the band of \( A (ku \geq 0) \).

- `ab`
  Array of size \( ldab \times n \).

  The matrix \( A \) in band storage, in rows \( kl+1 \) to \( 2kl+ku+1 \); rows 1 to \( kl \) of the matrix need not be set. The \( j \)-th column of \( A \) is stored in the array `ab` as follows: \( ab[kl+ku+i+j*(j-1)*ldab] = A(i,j) \) for \( \max(1,j-ku) \leq i \leq \min(m,j+kl) \).
ldab

The leading dimension of the array ab.

(ldab ≥ 2kl + ku + 1)

Output Parameters

ab

On exit, details of the factorization: U is stored as an upper triangular band matrix with kl+ku superdiagonals in rows 1 to kl+ku+1, and the multipliers used during the factorization are stored in rows kl+ku+2 to 2*kl+ku+1. See the Application Notes below for further details.

info

= 0: successful exit
< 0: if info = -i, the i-th argument had an illegal value,
> 0: if info = +i, the matrix element U(i,i) is 0. The factorization has been completed, but the factor U is exactly singular. Division by 0 will occur if you use the factor U for solving a system of linear equations.

Application Notes

The band storage scheme is illustrated by the following example, when m = n = 6, kl = 2, ku = 1:

<table>
<thead>
<tr>
<th></th>
<th>a11</th>
<th>a12</th>
<th>a13</th>
<th>a14</th>
<th>a15</th>
<th>a16</th>
</tr>
</thead>
<tbody>
<tr>
<td>a11</td>
<td>*</td>
<td>a22</td>
<td>a23</td>
<td>a24</td>
<td>a25</td>
<td>a26</td>
</tr>
</tbody>
</table>
| a21 | a32 | a33 | a34 | a35 | a36 |   *
| a31 | a42 | a43 | a44 | a45 | a46 |   *
| a41 |     | a53 | a54 | a55 | a56 |   *
| a51 |     |     | a64 | a65 | a66 |   *

The function does not use array elements marked *; elements marked + need not be set on entry, but the function requires them to store elements of U, because of fill-in resulting from the row interchanges.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

?dbtrf

Computes an LU factorization of a general band matrix with no pivoting (local blocked algorithm).

Syntax

void sdbtrf (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, float *ab, MKL_INT *ldb, MKL_INT *info);

void ddbtrf (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, double *ab, MKL_INT *ldb, MKL_INT *info);

void cdbtrf (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, MKL_Complex8 *ab, MKL_INT *ldb, MKL_INT *info);

void zdbtrf (MKL_INT *m, MKL_INT *n, MKL_INT *kl, MKL_INT *ku, MKL_Complex16 *ab, MKL_INT *ldb, MKL_INT *info);

Include Files

• mkl_scalapack.h
Description
This function computes an LU factorization of a real \( m \)-by-\( n \) band matrix \( A \) without using partial pivoting or row interchanges.
This is the blocked version of the algorithm, calling BLAS Routines and Functions.

Input Parameters

\( m \)
The number of rows of the matrix \( A \) (\( m \geq 0 \)).

\( n \)
The number of columns in \( A \)(\( n \geq 0 \)).

\( kl \)
The number of sub-diagonals within the band of \( A \) (\( kl \geq 0 \)).

\( ku \)
The number of super-diagonals within the band of \( A \) (\( ku \geq 0 \)).

\( ab \)
Array of size \( ldab \) * \( n \).
The matrix \( A \) in band storage, in rows \( kl+1 \) to \( 2kl+ku+1 \); rows 1 to \( kl \) need not be set. The \( j \)-th column of \( A \) is stored in the array \( ab \) as follows:
\[
ab[(kl + ku + j + (j-1) \times ldab)] = A(i, j) \text{ for } \max(1, j - ku) \leq i \leq \min(m, j + kl).
\]

\( ldab \)
The leading dimension of the array \( ab \).
(\( ldab \geq 2kl + ku + 1 \))

Output Parameters

\( ab \)
On exit, details of the factorization: \( U \) is stored as an upper triangular band matrix with \( kl + ku \) superdiagonals in rows 1 to \( kl + ku + 1 \), and the multipliers used during the factorization are stored in rows \( kl + ku + 2 \) to \( 2 \times kl + ku + 1 \). See the Application Notes below for further details.

\( info \)
\[ = 0: \text{successful exit} \]
\[ < 0: \text{if } info = -i, \text{the } i\text{-th argument had an illegal value,} \]
\[ > 0: \text{if } info = +i, \text{the matrix element } U(i, i) \text{ is 0. The factorization has been completed, but the factor } U \text{ is exactly singular. Division by 0 will occur if you use the factor } U \text{ for solving a system of linear equations.} \]

Application Notes
The band storage scheme is illustrated by the following example, when \( m = n = 6, kl = 2, ku = 1 \):

\[
\begin{array}{cccccc}
\ast & a_{12} & a_{33} & a_{44} & a_{55} & a_{66} \\
a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\
a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & \ast \\
a_{31} & a_{42} & a_{53} & a_{64} & \ast & \ast
\end{array}
\]

\[
\begin{array}{cccccc}
\ast & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\
u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\
m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & \ast \\
m_{31} & m_{42} & m_{53} & m_{64} & \ast & \ast
\end{array}
\]

The function does not use array elements marked \( \ast \).

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
Computes an LU factorization of a general tridiagonal matrix with no pivoting (local blocked algorithm).

Syntax

void sdttrf (MKL_INT *n, float *dl, float *d, float *du, MKL_INT *info);
void ddttrf (MKL_INT *n, double *dl, double *d, double *du, MKL_INT *info);
void cdttrf (MKL_INT *n, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_INT *info);
void zdttrf (MKL_INT *n, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

The ?dttrf function computes an LU factorization of a real or complex tridiagonal matrix A using elimination without partial pivoting.

The factorization has the form $A = L*U$, where $L$ is a product of unit lower bidiagonal matrices and $U$ is upper triangular with nonzeros only in the main diagonal and first superdiagonal.

Input Parameters

$n$  
The order of the matrix $A$ ($n \geq 0$).

dl, d, du  
Arrays containing elements of $A$.

The array $dl$ of size $(n-1)$ contains the sub-diagonal elements of $A$.

The array $d$ of size $n$ contains the diagonal elements of $A$.

The array $du$ of size $(n-1)$ contains the super-diagonal elements of $A$.

Output Parameters

$dl$  
Overwritten by the $(n-1)$ multipliers that define the matrix $L$ from the LU factorization of $A$.

d  
Overwritten by the $n$ diagonal elements of the upper triangular matrix $U$ from the LU factorization of $A$.

$du$  
Overwritten by the $(n-1)$ elements of the first super-diagonal of $U$.

$info$  
= 0: successful exit

< 0: if $info = -i$, the $i$-th argument had an illegal value,

> 0: if $info = i$, the matrix element $U(i,i)$ is exactly 0. The factorization has been completed, but the factor $U$ is exactly singular. Division by 0 will occur if you use the factor $U$ for solving a system of linear equations.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.
**?dttrsv**

Solves a general tridiagonal system of linear equations using the LU factorization computed by ?dttrf.

**Syntax**

```c
void sdttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, float *dl, float *d, float *du, float *b, MKL_INT *ldb, MKL_INT *info);
void ddttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, double *dl, double *d, double *du, double *b, MKL_INT *ldb, MKL_INT *info);
void cdttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *dl, MKL_Complex8 *d, MKL_Complex8 *du, MKL_Complex8 *b, MKL_INT *ldb, MKL_INT *info);
void zdttrsv(char *uplo, char *trans, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *dl, MKL_Complex16 *d, MKL_Complex16 *du, MKL_Complex16 *b, MKL_INT *ldb, MKL_INT *info);
```

**Include Files**

- mkl_scalapack.h

**Description**

The ?dttrsv function solves one of the following systems of linear equations:

\[
LX = B, \quad L^T X = B, \quad \text{or} \quad L^H X = B, \\
UX = B, \quad U^T X = B, \quad \text{or} \quad U^H X = B
\]

with factors of the tridiagonal matrix \( A \) from the LU factorization computed by ?dttrf.

**Input Parameters**

- **uplo**
  Specifies whether to solve with \( L \) or \( U \).

- **trans**
  Must be 'N' or 'T' or 'C'.

  Indicates the form of the equations:

  - If \( \text{trans} = 'N' \), then \( A X = B \) is solved for \( X \) (no transpose).
  - If \( \text{trans} = 'T' \), then \( A^T X = B \) is solved for \( X \) (transpose).
  - If \( \text{trans} = 'C' \), then \( A^H X = B \) is solved for \( X \) (conjugate transpose).

- **n**
  The order of the matrix \( A (n \geq 0) \).

- **nrhs**
  The number of right-hand sides, that is, the number of columns in the matrix \( B (nrhs \geq 0) \).

- **dl, d, du, b**
  The array \( dl \) of size \((n - 1)\) contains the \((n - 1)\) multipliers that define the matrix \( L \) from the \( LU \) factorization of \( A \).
  The array \( d \) of size \( n \) contains \( n \) diagonal elements of the upper triangular matrix \( U \) from the \( LU \) factorization of \( A \).
  The array \( du \) of size \((n - 1)\) contains the \((n - 1)\) elements of the first super-diagonal of \( U \).
On entry, the array \( b \) of size \( ldb \times nrhs \) contains the right-hand side of matrix \( B \).

\[ ldb \]
The leading dimension of the array \( b \); \( ldb \geq \max(1, n) \).

**Output Parameters**

\( b \) Overwritten by the solution matrix \( X \).

\( info \)
- \( info = 0 \), the execution is successful.
- \( info = -i \), the \( i \)-th parameter had an illegal value.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

?pttrsv

Solves a symmetric (Hermitian) positive-definite tridiagonal system of linear equations, using the \( L*D*L^H \) factorization computed by ?pttrf.

**Syntax**

void spttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , float *d , float *e , float *b , MKL_INT *ldb , MKL_INT *info );

void dpttrsv (char *trans , MKL_INT *n , MKL_INT *nrhs , double *d , double *e , double *b , MKL_INT *ldb , MKL_INT *info );

void cpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , float *d , MKL_Complex8 *e , MKL_Complex8 *b , MKL_INT *ldb , MKL_INT *info );

void zpttrsv (char *uplo , char *trans , MKL_INT *n , MKL_INT *nrhs , double *d , MKL_Complex16 *e , MKL_Complex16 *b , MKL_INT *ldb , MKL_INT *info );

**Include Files**

- mkl_scalapack.h

**Description**

The ?pttrsv function solves one of the triangular systems:

\[
L^T X = B, \text{ or } L X = B \] for real flavors,

or

\[
L^* X = B, \text{ or } L^{H*} X = B, \]

\[
U^* X = B, \text{ or } U^{H*} X = B \] for complex flavors,

where \( L \) (or \( U \) for complex flavors) is the Cholesky factor of a Hermitian positive-definite tridiagonal matrix \( A \) such that

\[
A = L^* D^* L^H \] (computed by spttrf/dpttrf)

or

\[
A = U^{H*} D^* U \text{ or } A = L^* D^* L^H \] (computed by cpttrf/zpttrf).


**Input Parameters**

*uplo*  
Must be 'U' or 'L'.

Specifies whether the superdiagonal or the subdiagonal of the tridiagonal matrix *A* is stored and the form of the factorization:

If *uplo* = 'U', *e* is the superdiagonal of *U*, and *A* = *U*H*D*U or *A* = *L*D*LU;

if *uplo* = 'L', *e* is the subdiagonal of *L*, and *A* = *L*D*LH.

The two forms are equivalent, if *A* is real.

*trans*  
Specifies the form of the system of equations:

for real flavors:

if *trans* = 'N': *L*X = *B* (no transpose)

if *trans* = 'T': *L*H*X = *B* (transpose)

for complex flavors:

if *trans* = 'N': *U*X = *B* or *L*X = *B* (no transpose)

if *trans* = 'C': *U*H*X = *B* or *L*H*X = *B* (conjugate transpose).

*n*  
The order of the tridiagonal matrix *A*. *n* ≥ 0.

*nrhs*  
The number of right hand sides, that is, the number of columns of the matrix *B*. *nrhs* ≥ 0.

*d*  
array of size *n*. The *n* diagonal elements of the diagonal matrix *D* from the factorization computed by ?pttrf.

*e*  
array of size (*n*-1). The (*n*-1) off-diagonal elements of the unit bidiagonal factor *U* or *L* from the factorization computed by ?pttrf. See *uplo*.

*b*  
array of size *ldb* *nrhs*.

On entry, the right hand side matrix *B*.

*ldb*  
The leading dimension of the array *b*.

*ldb* ≥ max(1, *n*).

**Output Parameters**

*b*  
On exit, the solution matrix *X*.

*info*  
= 0: successful exit

< 0: if *info* = −*i*, the *i*-th argument had an illegal value.

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.
?steqr2
Computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method.

Syntax

void ssteqr2 (char *compz, MKL_INT *n, float *d, float *e, float *z, MKL_INT *ldz, 
MKL_INT *nr, float *work, MKL_INT *info);
void dsteqr2 (char *compz, MKL_INT *n, double *d, double *e, double *z, MKL_INT 
*ldz, MKL_INT *nr, double *work, MKL_INT *info);

Include Files

• mkl_scalapack.h

Description

The ?steqr2 function is a modified version of LAPACK function ?steqr. The ?steqr2 function computes all eigenvalues and, optionally, eigenvectors of a symmetric tridiagonal matrix using the implicit QL or QR method. ?steqr2 is modified from ?steqr to allow each ScaLAPACK process running ?steqr2 to perform updates on a distributed matrix Q. Proper usage of ?steqr2 can be gleaned from examination of ScaLAPACK function p?syev.

Input Parameters

compz

Must be 'N' or 'I'.

If compz = 'N', the function computes eigenvalues only. If compz = 'I', the function computes the eigenvalues and eigenvectors of the tridiagonal matrix $T$.

$z$ must be initialized to the identity matrix by p?laset or ?laset prior to entering this function.

n

The order of the matrix $T (n \geq 0)$.

d, e, work

Arrays:

d contains the diagonal elements of $T$. The size of d must be at least max(1, n).

e contains the (n-1) subdiagonal elements of $T$. The size of e must be at least max(1, n-1).

work is a workspace array. The size of work is max(1, 2*n-2). If compz = 'N', then work is not referenced.

z

(local)

Array of global size n* n and of local size ldz* nr.

If compz = 'V', then z contains the orthogonal matrix used in the reduction to tridiagonal form.

ldz

The leading dimension of the array z. Constraints:

$ldz \geq 1,$
ldz ≥ max(1, n), if eigenvectors are desired.

nr = max(1, numroc(n, nb, myprow, 0, nprocs)).

If compz = 'N', then nr is not referenced.

Output Parameters

d
On exit, the eigenvalues in ascending order, if info = 0.
See also info.

e
On exit, e has been destroyed.

z
On exit, if info = 0, then,
if compz = 'V', z contains the orthonormal eigenvectors of the original symmetric matrix, and if compz = 'I', z contains the orthonormal eigenvectors of the symmetric tridiagonal matrix. If compz = 'N', then z is not referenced.

info
info = 0, the exit is successful.
info < 0: if info = -i, the i-th had an illegal value.
info > 0: the algorithm has failed to find all the eigenvalues in a total of 30n iterations;
if info = i, then i elements of e have not converged to zero; on exit, d and e contain the elements of a symmetric tridiagonal matrix, which is orthogonally similar to the original matrix.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

?trmvt
Performs matrix-vector operations.

Syntax
void strmvt (const char* uplo, const MKL_INT* n, const float* t, const MKL_INT* ldt, float* x, const MKL_INT* incx, const float* y, const MKL_INT* incy, float* w, const MKL_INT* incw, const float* z, const MKL_INT* incz);

void dtrmvt (const char* uplo, const MKL_INT* n, const double* t, const MKL_INT* ldt, double* x, const MKL_INT* incx, const double* y, const MKL_INT* incy, double* w, const MKL_INT* incw, const double* z, const MKL_INT* incz);

void ctrmvt (const char* uplo, const MKL_INT* n, const MKL_Complex8* t, const MKL_INT* ldt, MKL_Complex8* x, const MKL_INT* incx, const MKL_Complex8* y, const MKL_INT* incy, MKL_Complex8* w, const MKL_INT* incw, const MKL_Complex8* z, const MKL_INT* incz);

void ztrmvt (const char* uplo, const MKL_INT* n, const MKL_Complex16* t, const MKL_INT* ldt, MKL_Complex16* x, const MKL_INT* incx, const MKL_Complex16* y, const MKL_INT* incy, MKL_Complex16* w, const MKL_INT* incw, const MKL_Complex16* z, const MKL_INT* incz);
Include Files

- mkl_scalapack.h

Description

?trmv performs the matrix-vector operations as follows:

strmv and dtrmv: \( x := T' *y \), and \( w := T *z \)

ctrmv and ztrmv: \( x := \text{conjg}( T' ) *y \), and \( w := T *z \),

where \( x \) is an \( n \) element vector and \( T \) is an \( n \)-by-\( n \) upper or lower triangular matrix.

Input Parameters

\texttt{uplo}

On entry, \texttt{uplo} specifies whether the matrix is an upper or lower triangular matrix as follows:

\texttt{uplo} = 'U' or 'u'

\( A \) is an upper triangular matrix.

\texttt{uplo} = 'L' or 'l'

\( A \) is a lower triangular matrix.

Unchanged on exit.

\texttt{n}

On entry, \( n \) specifies the order of the matrix \( A \). \( n \) must be at least zero.

Unchanged on exit.

\texttt{t}

Array of size ( \texttt{ldt}, \texttt{n} ).

Before entry with \texttt{uplo} = 'U' or 'u', the leading \( n \)-by-\( n \) upper triangular part of the array \( t \) must contain the upper triangular matrix and the strictly lower triangular part of \( t \) is not referenced.

Before entry with \texttt{uplo} = 'L' or 'l', the leading \( n \)-by-\( n \) lower triangular part of the array \( t \) must contain the lower triangular matrix and the strictly upper triangular part of \( t \) is not referenced.

\texttt{ldt}

On entry, \texttt{lda} specifies the first dimension of \( A \) as declared in the calling (sub) program. \texttt{lda} must be at least max( 1, \texttt{n} ).

Unchanged on exit.

\texttt{incx}

On entry, \texttt{incx} specifies the increment for the elements of \( x \). \texttt{incx} must not be zero.

Unchanged on exit.

\texttt{y}

Array of size at least ( 1 + ( \texttt{n} - 1 )*abs( \texttt{incy} ) ).

Before entry, the incremented array \( y \) must contain the \( n \) element vector \( y \).

Unchanged on exit.

\texttt{incy}

On entry, \texttt{incy} specifies the increment for the elements of \( y \). \texttt{incy} must not be zero.

Unchanged on exit.
On entry, \( \text{incw} \) specifies the increment for the elements of \( w \). \( \text{incw} \) must not be zero.
Unchanged on exit.

- **z**
  - Array of size at least \( 1 + (n - 1) \times \text{abs}(\text{incz}) \).
  - Before entry, the incremented array \( z \) must contain the \( n \) element vector \( z \).
  - Unchanged on exit.

- **incz**
  - On entry, \( \text{incz} \) specifies the increment for the elements of \( z \). \( \text{incz} \) must not be zero.
  - Unchanged on exit.

**Output Parameters**

- **t**
  - Before entry with \( \text{uplo} = 'U' \) or \( 'u' \), the leading \( n \)-by-\( n \) upper triangular part of the array \( t \) must contain the upper triangular matrix and the strictly lower triangular part of \( t \) is not referenced.
  - Before entry with \( \text{uplo} = 'L' \) or \( 'l' \), the leading \( n \)-by-\( n \) lower triangular part of the array \( t \) must contain the lower triangular matrix and the strictly upper triangular part of \( t \) is not referenced.

- **x**
  - Array of size at least \( 1 + (n - 1) \times \text{abs}(\text{incx}) \).
  - On exit, \( x = T^* y \).

- **w**
  - Array of size at least \( 1 + (n - 1) \times \text{abs}(\text{incw}) \).
  - On exit, \( w = T^* z \).

**pilaenv**

*Returns the positive integer value of the logical blocking size.*

**Syntax**

```c
MKL_INT pilaenv (const MKL_INT *ictxt, const char *prec);
```

**Include Files**

- mkl_pblas.h

**Description**

\( \text{pilaenv} \) returns the positive integer value of the logical blocking size. This value is machine and precision specific. This version provides a logical blocking size which should give good though not optimal performance on many of the currently available distributed-memory concurrent computers. You are encouraged to modify this subroutine to set this tuning parameter for your particular machine.

**Input Parameters**

- **ictxt**
  - On entry, \( \text{ictxt} \) specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of \( \text{ictxt} \) is local.
prec

On input, prec specifies the precision for which the logical block size should be returned as follows:

- prec = 'S' or 's' single precision real,
- prec = 'D' or 'd' double precision real,
- prec = 'C' or 'c' single precision complex,
- prec = 'Z' or 'z' double precision complex,
- prec = 'I' or 'i' integer.

Application Notes

Before modifying this routine to tune the library performance on your system, be aware of the following:

1. The value this function returns must be strictly larger than zero,
2. If you are planning to link your program with different instances of the library (for example, on a heterogeneous machine), you must compile each instance of the library with exactly the same version of this routine for obvious interoperability reasons.

pilaenvx

Called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment.

Syntax

MKL_INT pilaenvx (const MKL_INT* ictxt, const MKL_INT* ispec, const char* name, const char* opts, const MKL_INT* n1, const MKL_INT* n2, const MKL_INT* n3, const MKL_INT*

Include Files

• mkl.h

Description

pilaenvx is called from the ScaLAPACK routines to choose problem-dependent parameters for the local environment. See ispec for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

ictxt

(local input) On entry, ictxt specifies the BLACS context handle, indicating the global context of the operation. The context itself is global, but the value of ictxt is local.

ispec

(global input)

Specifies the parameter to be returned as the value of pilaenvx.

- = 1: the optimal blocksize; if this value is 1, an unblocked algorithm will give the best performance (unlikely).
- = 2: the minimum block size for which the block routine should be used; if the usable block size is less than this value, an unblocked routine should be used.
= 3: the crossover point (in a block routine, for N less than this value, an unblocked routine should be used).

= 4: the number of shifts, used in the nonsymmetric eigenvalue routines (DEPRECATED).

= 5: the minimum column dimension for blocking to be used; rectangular blocks must have dimension at least \( k \) by \( m \), where \( k \) is given by \( \text{pilaenvx}(2,...) \) and \( m \) by \( \text{pilaenvx}(5,...) \).

= 6: the crossover point for the SVD (when reducing an \( m \) by \( n \) matrix to bidiagonal form, if \( \max(m,n)/\min(m,n) \) exceeds this value, a QR factorization is used first to reduce the matrix to a triangular form).

= 7: the number of processors.

= 8: the crossover point for the multishift QR method for nonsymmetric eigenvalue problems (DEPRECATED).

= 9: maximum size of the subproblems at the bottom of the computation tree in the divide-and-conquer algorithm (used by \?gelsd and \?geadd).

=10: IEEE NaN arithmetic can be trusted not to trap.

=11: infinity arithmetic can be trusted not to trap.

12 <= \textit{ispec} <= 16:

\textit{p?hseqr} or one of its subroutines, see piparmq for detailed explanation.

17 <= \textit{ispec} <= 22:

Parameters for \textit{pb?trord/p?hseqr} (not all), as follows:

=17: maximum number of concurrent computational windows;

=18: number of eigenvalues/bulges in each window;

=19: computational window size;

=20: minimal percentage of FLOPS required for performing matrix-matrix multiplications instead of pipelined orthogonal transformations;

=21: width of block column slabs for row-wise application of pipelined orthogonal transformations in their factorized form;

=22: the maximum number of eigenvalues moved together over a process border;

=23: the number of processors involved in Aggressive Early Deflation (AED);

=99: Maximum iteration chunksize in OpenMP parallelization.

\textbf{name}

\textit{name} (global input)

The name of the calling subroutine, in either upper case or lower case.

\textbf{opts}

\textit{opts} (global input) The character options to the subroutine name, concatenated into a single character string. For example, \textit{uplo} = 'U', \textit{trans} = 'T', and \textit{diag} = 'N' for a triangular routine would be specified as \textit{opts} = 'UTN'.

\textbf{n1, n2, n3, and n4}

\textit{n1, n2, n3, and n4} (global input) Problem dimensions for the subroutine name; these may not all be required.
Output Parameters

result (global output)

>= 0: the value of the parameter specified by ispec.

< 0: if pilaenvx = -k, the k-th argument had an illegal value.

Application Notes

The following conventions have been used when calling ilaenv from the LAPACK routines:

1. opts is a concatenation of all of the character options to subroutine name, in the same order that they appear in the argument list for name, even if they are not used in determining the value of the parameter specified by ispec.

2. The problem dimensions n1, n2, n3, and n4 are specified in the order that they appear in the argument list for name. n1 is used first, n2 second, and so on, and unused problem dimensions are passed a value of -1.

3. The parameter value returned by ilaenv is checked for validity in the calling subroutine. For example, ilaenv is used to retrieve the optimal block size for strtri as follows:

   NB = ilaenv( 1, 'STRTRI', UPLO // DIAG, N, -1, -1, -1 );
   if( NB<=1 ) {
       NB = MAX( 1, N );
   }

The same conventions hold for this ScaLAPACK-style variant.

pjlaenv

Called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment.

Syntax

MKL_INT pjlaenv (const MKL_INT*ictxt, const MKL_INT*ispec, const char*name, const char*opts, const MKL_INT*n1, const MKL_INT*n2, const MKL_INT*n3, const MKL_INT*n4);

Include Files

• mkl.h

Description

pjlaenv is called from the ScaLAPACK symmetric and Hermitian tailored eigen-routines to choose problem-dependent parameters for the local environment. See ispec for a description of the parameters. This version provides a set of parameters which should give good, though not optimal, performance on many of the currently available computers. You are encouraged to modify this subroutine to set the tuning parameters for your particular machine using the option and problem size information in the arguments.

Input Parameters

ispec (global input) Specifies the parameter to be returned as the value of pjlaenv.

= 1: the data layout blocksize;

= 2: the panel blocking factor;
= 3: the algorithmic blocking factor;
= 4: execution path control;
= 5: maximum size for direct call to the LAPACK routine.

name

(global input) The name of the calling subroutine, in either upper case or
lower case.

opts

(global input) The character options to the subroutine name, concatenated
into a single character string. For example, uplo = 'U', trans = 'T',
and diag = 'N' for a triangular routine would be specified as opts = 'UTN'.

n1, n2, n3, and n4

(global input) Problem dimensions for the subroutine name; these may not
all be required. At present, only n1 is used, and it (n1) is used only for
'TTRD'.

Output Parameters

result

(global or local output)

>= 0: the value of the parameter specified by ispec.

< 0: if pjlaenv = -k, the k-th argument had an illegal value. Most
parameters set via a call to pjlaenv must be identical on all
processors and hence pjlaenv will return the same value to all
processors (i.e. global output). However some, in particular, the panel
blocking factor can be different on each processor and hence pjlaenv
can return different values on different processors (i.e. local output).

Application Notes

The following conventions have been used when calling pjlaenv from the ScaLAPACK routines:

1. opts is a concatenation of all of the character options to subroutine name, in the same order that they
appear in the argument list for name, even if they are not used in determining the value of the
parameter specified by ispec.

2. The problem dimensions n1, n2, n3, and n4 are specified in the order that they appear in the argument
list for name. n1 is used first, n2 second, and so on, and unused problem dimensions are passed a
value of -1.

   a. The parameter value returned by pjlaenv is checked for validity in the calling subroutine. For
      example, pjlaenv is used to retrieve the optimal blocksize for STRTRI as follows:

      ```
      NB = pjlaenv( 1, 'STRTRI', UPL0 // DIAG, N, -1, -1, -1 );
      IF( NB>=1 ) {
          NB = MAX( 1, N );
      }
      ```

pjlaenv is patterned after ilaenv and keeps the same interface in anticipation of future needs, even though
pjlaenv is only sparsely used at present in ScaLAPACK. Most ScaLAPACK codes use the input data layout
blocking factor as the algorithmic blocking factor - hence there is no need or opportunity to set the
algorithmic or data decomposition blocking factor. pXYTevx.f and pXYTgvx.f and pXYTtrd.f are the
only codes which call pjlaenv. pXYTevx.f and pXYTgvx.f redistribute the data to the best data layout
for each transformation. pXYTtrd.f uses a data layout blocking factor of 1.
Additional ScALAPACK Routines

void pchettrd (const char *uplo, const MKL_INT *n, MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, float *d, float *e, MKL_Complex8 *tau, MKL_Complex8 *work, const MKL_INT *iwork, MKL_INT *info);

void pzhetttrd (const char *uplo, const MKL_INT *n, MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, double *d, double *e, MKL_Complex16 *tau, MKL_Complex16 *work, const MKL_INT *iwork, MKL_INT *info);

void pslaed0 (const MKL_INT *n, float *d, float *e, float *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *descq, float *work, MKL_INT *iwork, MKL_INT *info);

void pslaed0 (const MKL_INT *n, double *d, double *e, double *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *descq, double *work, MKL_INT *iwork, MKL_INT *info);

void pslaed1 (const MKL_INT *n, MKL_INT *n1, float *d, const MKL_INT *id, float *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *descq, const float *rho, float *work, MKL_INT *iwork, MKL_INT *info);

void pslaed1 (const MKL_INT *n, const MKL_INT *n1, double *d, const MKL_INT *id, double *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *descq, const double *rho, double *work, MKL_INT *iwork, MKL_INT *info);

void pslaed2 (const MKL_INT *ictxt, MKL_INT *k, const MKL_INT *n, const MKL_INT *n1, const MKL_INT *nb, float *d, const MKL_INT *drow, const MKL_INT *dcol, float *q, const MKL_INT *ldq, float *rho, const float *z, float *w, float *dlamda, float *q2, const MKL_INT *ldq2, float *qbuf, MKL_INT *ctot, MKL_INT *psm, const MKL_INT *npcol, MKL_INT *indx, MKL_INT *inde, MKL_INT *indcol, MKL_INT *coltyp, MKL_INT *nn, MKL_INT *nn1, MKL_INT *nn2, MKL_INT *ib1, MKL_INT *ib2);

void pslaed3 (const MKL_INT *ictxt, MKL_INT *k, const MKL_INT *n, const MKL_INT *n1, const MKL_INT *nb, float *d, const MKL_INT *drow, const MKL_INT *dcol, float *q, const MKL_INT *ldq, float *rho, float *dlamda, float *w, const float *z, float *u, const MKL_INT *ldu, float *buf, MKL_INT *indx, MKL_INT *inde, MKL_INT *indcol, MKL_INT *indrow, MKL_INT *inde, MKL_INT *ctot, const MKL_INT *npcol, MKL_INT *info);

void pslaed3 (const MKL_INT *ictxt, MKL_INT *k, const MKL_INT *n, const MKL_INT *n1, const MKL_INT *nb, double *d, const MKL_INT *drow, const MKL_INT *dcol, double *rho, double *dlamda, double *w, const double *z, double *u, const MKL_INT *ldu, double *buf, MKL_INT *indx, MKL_INT *inde, MKL_INT *indcol, MKL_INT *indrow, MKL_INT *inde, MKL_INT *ctot, const MKL_INT *npcol, MKL_INT *info);

void pslaedz (const MKL_INT *n, const MKL_INT *n1, const MKL_INT *id, const float *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *ldq, const MKL_INT *descq, float *z, float *work);

void pslaedz (const MKL_INT *n, const MKL_INT *n1, const MKL_INT *id, const double *q, const MKL_INT *iq, const MKL_INT *jq, const MKL_INT *ldq, const MKL_INT *descq, double *z, double *work);

void pdlaectb (const double *sigma, const MKL_INT *n, const double *d, MKL_INT *count);
This section describes ScaLAPACK utility functions and routines. Summary information about these routines is given in the following table:

For descriptions of these functions, please see http://www.netlib.org/scalapack/explore-html/files.html.
### ScaLAPACK Utility Functions and Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p?labad</td>
<td>s,d</td>
<td>Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.</td>
</tr>
<tr>
<td>p?lachkieee</td>
<td>s,d</td>
<td>Performs a simple check for the features of the IEEE standard.</td>
</tr>
<tr>
<td>p?lamch</td>
<td>s,d</td>
<td>Determines machine parameters for floating-point arithmetic.</td>
</tr>
<tr>
<td>p?lasnbt</td>
<td>s,d</td>
<td>Computes the position of the sign bit of a floating-point number.</td>
</tr>
<tr>
<td>descinit</td>
<td>N/A</td>
<td>Initializes the array descriptor for distributed matrix.</td>
</tr>
<tr>
<td>numroc</td>
<td>N/A</td>
<td>Computes the number of rows or columns of a distributed matrix owned by the process.</td>
</tr>
</tbody>
</table>

### See Also
- p?lachkieee  | Performs a simple check for the features of the IEEE standard. |
- numroc       | Computes the number of rows or columns of a distributed matrix owned by the process. |

### p?labad

**Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.**

**Syntax**

```c
void pslabad (MKL_INT *ictxt, float *small, float *large);
void pdlabad (MKL_INT *ictxt, double *small, double *large);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?labad` function takes as input the values computed by `p?lamch` for underflow and overflow, and returns the square root of each of these values if the log of `large` is sufficiently large. This function is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by `p?lamch`. This function is needed because `p?lamch` does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this function performs a global minimization and maximization on these values, to support heterogeneous computing networks.

**Input Parameters**

- `ictxt` (global)
  - The BLACS context handle in which the computation takes place.
- `small` (local)
  - On entry, the underflow threshold as computed by `p?lamch`.
- `large` (local)
  - On entry, the overflow threshold as computed by `p?lamch`. 

Excerpt from the document:

```
See Also
pxerbla  Error handling routine called by ScaLAPACK routines.

p?labad
Returns the square root of the underflow and overflow thresholds if the exponent-range is very large.

Syntax
void pslabad (MKL_INT *ictxt, float *small, float *large);
void pdlabad (MKL_INT *ictxt, double *small, double *large);

Include Files
- mkl_scalapack.h

Description
The p?labad function takes as input the values computed by p?lamch for underflow and overflow, and returns the square root of each of these values if the log of large is sufficiently large. This function is intended to identify machines with a large exponent range, such as the Crays, and redefine the underflow and overflow limits to be the square roots of the values computed by p?lamch. This function is needed because p?lamch does not compensate for poor arithmetic in the upper half of the exponent range, as is found on a Cray.

In addition, this function performs a global minimization and maximization on these values, to support heterogeneous computing networks.

Input Parameters

ictxt  (global)
   - The BLACS context handle in which the computation takes place.

small  (local).
   - On entry, the underflow threshold as computed by p?lamch.

large  (local).
   - On entry, the overflow threshold as computed by p?lamch.
```
Output Parameters

small (local).
On exit, if $\log_{10}(\text{large})$ is sufficiently large, the square root of small, otherwise unchanged.

large (local).
On exit, if $\log_{10}(\text{large})$ is sufficiently large, the square root of large, otherwise unchanged.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.

p?lachkieee
Performs a simple check for the features of the IEEE standard.

Syntax

void pslachkieee (MKL_INT *isieee , float *rmax , float *rmin );
void pdlachkieee (MKL_INT *isieee , float *rmax , float *rmin );

Include Files

• mkl_scalapack.h

Description

The p?lachkieee function performs a simple check to make sure that the features of the IEEE standard are implemented. In some implementations, p?lachkieee may not return.

This is a ScaLAPACK internal function and arguments are not checked for unreasonable values.

Input Parameters

rmax (local).
The overflow threshold ($= ?\text{lamch} ('O')$).

rmin (local).
The underflow threshold ($= ?\text{lamch} ('U')$).

Output Parameters

isieee (local).
On exit, $\text{isieee} = 1$ implies that all the features of the IEEE standard that we rely on are implemented. On exit, $\text{isieee} = 0$ implies that some the features of the IEEE standard that we rely on are missing.

See Also
Overview for details of ScaLAPACK array descriptor structures and related notations.
**p?lamch**  
*Determines machine parameters for floating-point arithmetic.*

**Syntax**

```c
float pslamch (MKL_INT *ictxt, char *cmach);
double pdlamch (MKL_INT *ictxt, char *cmach);
```

**Include Files**

- mkl_scalapack.h

**Description**

The `p?lamch` function determines single precision machine parameters.

**Input Parameters**

- **ictxt** (global). The BLACS context handle in which the computation takes place.
- **cmach** (global). Specifies the value to be returned by `p?lamch`:
  - `'E'` or `'e'`, `p?lamch := eps`
  - `'S'` or `'s'`, `p?lamch := sfmin`
  - `'B'` or `'b'`, `p?lamch := base`
  - `'P'` or `'p'`, `p?lamch := eps*base`
  - `'N'` or `'n'`, `p?lamch := t`
  - `'R'` or `'r'`, `p?lamch := rnd`
  - `'M'` or `'m'`, `p?lamch := emin`
  - `'U'` or `'u'`, `p?lamch := rmin`
  - `'L'` or `'l'`, `p?lamch := emax`
  - `'O'` or `'o'`, `p?lamch := rmax`,

where
- `eps` = relative machine precision
- `sfmin` = safe minimum, such that `1/sfmin` does not overflow
- `base` = base of the machine
- `prec` = `eps*base`
- `t` = number of (base) digits in the mantissa
- `rnd` = 1.0 when rounding occurs in addition, 0.0 otherwise
- `emin` = minimum exponent before (gradual) underflow
- `rmin` = underflow threshold - `base`(emin-1)
- `emax` = largest exponent before overflow
\[ r_{\text{max}} = \text{overflow threshold} - (b_{\text{ase}^{\text{max}}})(1-\text{eps}) \]

**Output Parameters**

**val**
Value returned by the function.

**See Also**

*Overview* for details of ScaLAPACK array descriptor structures and related notations.

**p?lasnbt**

*Computes the position of the sign bit of a floating-point number.*

**Syntax**

```c
void pslasnbt (MKL_INT *ieflag);
void pdlasnbt (MKL_INT *ieflag);
```

**Include Files**

- `mkl_scalapack.h`

**Description**

The `p?lasnbt` function finds the position of the sign bit of a single/double precision floating point number. This function assumes IEEE arithmetic, and hence, tests only the 32-nd bit (for single precision) or 32-nd and 64-th bits (for double precision) as a possibility for the sign bit. `sizeof(int)` is assumed equal to 4 bytes.

If a compile time flag (`NO_IEEE`) indicates that the machine does not have IEEE arithmetic, `ieflag = 0` is returned.

**Output Parameters**

**ieflag**

This flag indicates the position of the sign bit of any single/double precision floating point number.

- `ieflag = 0`, if the compile time flag `NO_IEEE` indicates that the machine does not have IEEE arithmetic, or if `sizeof(int)` is different from 4 bytes.
- `ieflag = 1` indicates that the sign bit is the 32-nd bit for a single precision function.
- In the case of a double precision function:
  - `ieflag = 1` indicates that the sign bit is the 32-nd bit (Big Endian).
  - `ieflag = 2` indicates that the sign bit is the 64-th bit (Little Endian).

**See Also**

*Overview* for details of ScaLAPACK array descriptor structures and related notations.

**descinit**

*Initializes the array descriptor for distributed matrix.*
**Syntax**

```c
void descinit (MKL_INT *desc, const MKL_INT *m, const MKL_INT *n, const MKL_INT *mb,
               const MKL_INT *nb, const MKL_INT *lrsrc, const MKL_INT *lcsrc, const MKL_INT *ictxt,
               const MKL_INT *lld, MKL_INT *info);
```

**Description**

The `descinit` function initializes the array descriptor for distributed matrix.

**Input Parameters**

- `desc` (global) array of dimension DLEN_. The array descriptor of a distributed matrix to be set.
- `m` (global input) The number of rows in the distributed matrix. M >= 0.
- `n` (global input) The number of columns in the distributed matrix. N >= 0.
- `mb` (global input) The blocking factor used to distribute the rows of the matrix. MB >= 1.
- `nb` (global input) The blocking factor used to distribute the columns of the matrix. NB >= 1.
- `lrsrc` (global input) The process row over which the first row of the matrix is distributed. 0 <= IRSRC < NPROW.
- `lcsrc` (global input) The process column over which the first column of the matrix is distributed. 0 <= ICSRC < NPCOL.
- `ictxt` (global input) The BLACS context handle, indicating the global context of the operation on the matrix. The context itself is global.
- `lld` (local input) The leading dimension of the local array storing the local blocks of the distributed matrix. LLD >= MAX(1,LOCr(M)). LOCr() denotes the number of rows of a global dense matrix that the process in a grid receives after data distributing.

**Output Parameters**

- `info` (output)
  - = 0: successful exit
  - < 0: if INFO = -i, the i-th argument had an illegal value

**See Also**

Overview for details of ScaLAPACK array descriptor structures and related notations.

**numroc**

*Computes the number of rows or columns of a distributed matrix owned by the process.*

**Syntax**

```c
MKL_INT numroc (const MKL_INT *n, const MKL_INT *nb, const MKL_INT *iproc, const MKL_INT *srcproc, const MKL_INT *nprocs);
```
Description

The `numroc` function computes the number of rows or columns of a distributed matrix owned by the process.

Input Parameters

- `n` (global input) The number of rows/columns in distributed matrix.
- `nb` (global input) Block size, size of the blocks the distributed matrix is split into.
- `iproc` (local input) The coordinate of the process whose local array row or column is to be determined.
- `srcproc` (global input) The coordinate of the process that possesses the first row or column of the distributed matrix.
- `nprocs` (global input) The total number processes over which the matrix is distributed.

Output Parameters

- `info` (output) Value returned by the function.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

ScaLAPACK Redistribution/Copy Routines

This section describes ScaLAPACK redistribution/copy routines. Summary information about these routines is given in the following table:

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p?gemr2d</td>
<td>s,d,c,z,i</td>
<td>Copies a submatrix from one general rectangular matrix to another.</td>
</tr>
<tr>
<td>p?trmr2d</td>
<td>s,d,c,z,i</td>
<td>Copies a submatrix from one trapezoidal matrix to another.</td>
</tr>
</tbody>
</table>

See Also

pxerbla Error handling routine called by ScaLAPACK routines.

p?gemr2d

Copies a submatrix from one general rectangular matrix to another.

Syntax

```c
void psgemr2d (MKL_INT *m, MKL_INT *n, float *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, float *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pdgemr2d (MKL_INT *m, MKL_INT *n, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pcgemr2d (MKL_INT *m, MKL_INT *n, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
```
void pzgemr2d (MKL_INT *m, MKL_INT *n, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pigemr2d (MKL_INT *m, MKL_INT *n, MKL_INT *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);

Include Files
• mkl_scalapack.h

Description
The p?gemr2d function copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With p?trmr2d, these functions are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context A is disjoint from context B. The general rules for which parameters need to be set are:
• All calling processes must have the correct m and n.
• Processes in context A must correctly define all parameters describing A.
• Processes in context B must correctly define all parameters describing B.
• Processes which are not members of context A must pass ctxt_a = -1 and need not set other parameters describing A.
• Processes which are not members of context B must pass ctxt_b = -1 and need not set other parameters describing B.

Because of its generality, p?gemr2d can be used for many operations not usually associated with copy functions. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this function can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this function requires an array descriptor with dtype_ = 1.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Input Parameters

- **m** (global) The number of rows of matrix A to be copied ($m \geq 0$).
- **n** (global) The number of columns of matrix A to be copied ($n \geq 0$).
- **a** (local) Pointer into the local memory to array of size $lld_a \times LOCc(ja+n-1)$ containing the source matrix A.
- **ia, ja** (global) The row and column indices in the array A indicating the first row and the first column, respectively, of the submatrix of A) to copy. $1 \leq ia \leq total_rows_in_a - m + 1$, $1 \leq ja \leq total_columns_in_a - n + 1$.
- **desca** (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix A.
  
  Only $dtype_\_a = 1$ is supported, so $dlen_\_ = 9$.
  
  If the calling process is not part of the context of A, $ctxt_\_a$ must be equal to -1.
- **ib, jb** (global) The row and column indices in the array B indicating the first row and the first column, respectively, of the submatrix B to which to copy the matrix. $1 \leq ib \leq total_rows_in_b - m + 1$, $1 \leq jb \leq total_columns_in_b - n + 1$.
- **descb** (global and local) array of size $dlen_\_$. The array descriptor for the distributed matrix B.
  
  Only $dtype_\_b = 1$ is supported, so $dlen_\_ = 9$.
  
  If the calling process is not part of the context of B, $ctxt_\_b$ must be equal to -1.
- **ictxt** (global).
  
  The context encompassing at least the union of all processes in context A and context B. All processes in the context $ictxt$ must call this function, even if they do not own a piece of either matrix.

Output Parameters

- **b** Pointer into the local memory to array of size $lld_b \times LOCc(jb+n-1)$.
  
  Overwritten by the submatrix from A.

See Also

Overview for details of ScaLAPACK array descriptor structures and related notations.

### p?trmr2d

Copies a submatrix from one trapezoidal matrix to another.

#### Syntax

```c
void pstrmr2d (char *uplo , char *diag , MKL_INT *m , MKL_INT *n , float *a , MKL_INT *ia , MKL_INT *ja , MKL_INT *desca , float *b , MKL_INT *ib , MKL_INT *jb , MKL_INT *descb , MKL_INT *ictxt );
```
void pdtrmr2d (char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, double *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, double *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pctrmr2d (char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, MKL_Complex8 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex8 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pztrmr2d (char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_INT *nrhs, MKL_Complex16 *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_Complex16 *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);
void pitrmr2d (char *uplo, char *diag, MKL_INT *m, MKL_INT *n, MKL_INT *a, MKL_INT *ia, MKL_INT *ja, MKL_INT *desca, MKL_INT *b, MKL_INT *ib, MKL_INT *jb, MKL_INT *descb, MKL_INT *ictxt);

Include Files
• mkl_scalapack.h

Description
The p?trmr2d function copies the indicated matrix or submatrix of A to the indicated matrix or submatrix of B. It provides a truly general copy from any block cyclicly-distributed matrix or submatrix to any other block cyclicly-distributed matrix or submatrix. With p?gemr2d, these functions are the only ones in the ScaLAPACK library which provide inter-context operations: they can take a matrix or submatrix A in context A (distributed over process grid A) and copy it to a matrix or submatrix B in context B (distributed over process grid B).

The p?trmr2d function assumes the matrix or submatrix to be trapezoidal. Only the upper or lower part is copied, and the other part is unchanged.

There does not need to be a relationship between the two operand matrices or submatrices other than their global size and the fact that they are both legal block cyclicly-distributed matrices or submatrices. This means that they can, for example, be distributed across different process grids, have varying block sizes and differing matrix starting points, or be contained in different sized distributed matrices.

Take care when context A is disjoint from context B. The general rules for which parameters need to be set are:
• All calling processes must have the correct m and n.
• Processes in context A must correctly define all parameters describing A.
• Processes in context B must correctly define all parameters describing B.
• Processes which are not members of context A must pass ctxt_a = -1 and need not set other parameters describing A.
• Processes which are not members of context B must pass ctxt_b = -1 and need not set other parameters describing B.

Because of its generality, p?trmr2d can be used for many operations not usually associated with copy functions. For instance, it can be used to take a matrix on one process and distribute it across a process grid, or the reverse. If a supercomputer is grouped into a virtual parallel machine with a workstation, for instance, this function can be used to move the matrix from the workstation to the supercomputer and back. In ScaLAPACK, it is called to copy matrices from a two-dimensional process grid to a one-dimensional process grid. It can be used to redistribute matrices so that distributions providing maximal performance can be used by various component libraries, as well.

Note that this function requires an array descriptor with dtype_ = 1.
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Input Parameters

uplo  (global) Specifies whether to copy the upper or lower part of the matrix or submatrix.

- uplo = 'U'  Copy the upper triangular part.
- uplo = 'L'  Copy the lower triangular part.

diag  (global) Specifies whether to copy the diagonal of the matrix or submatrix.

- diag = 'U'  Do not copy the diagonal.
- diag = 'N'  Copy the diagonal.

m  (global) The number of rows of matrix A to be copied (m≥0).

n  (global) The number of columns of matrix A to be copied (n≥0).

a  (local)
Pointer into the local memory to array of size lld_a* LOCc(ja+n-1) containing the source matrix A.

ia, ja  (global) The row and column indices in the array A indicating the first row and the first column, respectively, of the submatrix of A) to copy. 1 ≤ ia ≤ total_rows_in_a - m + 1, 1 ≤ ja ≤ total_columns_in_a - n + 1.

desca  (global and local) array of size dlen_. The array descriptor for the distributed matrix A.

Only dtype_a = 1 is supported, so dlen_ = 9.

If the calling process is not part of the context of A, ctxt_a must be equal to -1.

ib, jb  (global) The row and column indices in the array B indicating the first row and the first column, respectively, of the submatrix B to which to copy the matrix. 1 ≤ ib ≤ total_rows_in_b - m + 1, 1 ≤ jb ≤ total_columns_in_b - n + 1.

descb  (global and local) array of size dlen_. The array descriptor for the distributed matrix B.

Only dtype_b = 1 is supported, so dlen_ = 9.

If the calling process is not part of the context of B, ctxt_b must be equal to -1.
ictxt

(global).

The context encompassing at least the union of all processes in context $A$
and context $B$. All processes in the context ictxt must call this function,
even if they do not own a piece of either matrix.

Output Parameters

$b$

Pointer into the local memory to array of size $ld_b \times \text{LOC}(jb+n-1)$.
Overwritten by the submatrix from $A$.

See Also

Overview  for details of ScaLAPACK array descriptor structures and related notations.
Sparse Solver Routines

Intel® Math Kernel Library (Intel® MKL) provides user-callable sparsesolver software to solve real or complex, symmetric, structurally symmetric or nonsymmetric, positive definite, indefinite or Hermitian square sparse linear system of algebraic equations.

The terms and concepts required to understand the use of the Intel® MKL sparse solver routines are discussed in the Appendix A “Linear Solvers Basics”. If you are familiar with linear sparse solvers and sparse matrix storage schemes, you can skip these sections and go directly to the interface descriptions.

This chapter describes
- the direct sparse solver based on PARDISO® which is referred to here as Intel MKL PARDISO;
- the alternative interface for the direct sparse solver which is referred to here as the DSS interface;
- iterative sparse solvers (ISS) based on the reverse communication interface (RCI);
- and two preconditioners based on the incomplete LU factorization technique.

Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface

This section describes the interface to the shared-memory multiprocessing parallel direct sparse solver known as the Intel® MKL PARDISO solver.

The Intel® MKL PARDISO package is a high-performance, robust, memory efficient, and easy to use software package for solving large sparse linear systems of equations on shared memory multiprocessors. The solver uses a combination of left- and right-looking Level-3 BLAS supernode techniques [Schenk00-2]. To improve sequential and parallel sparse numerical factorization performance, the algorithms are based on a Level-3 BLAS update and pipelining parallelism is used with a combination of left- and right-looking supernode techniques [Schenk00, Schenk01, Schenk02, Schenk03]. The parallel pivoting methods allow complete supernode pivoting to compromise numerical stability and scalability during the factorization process. For sufficiently large problem sizes, numerical experiments demonstrate that the scalability of the parallel algorithm is nearly independent of the shared-memory multiprocessing architecture.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

The following table lists the names of the Intel® MKL PARDISO routines and describes their general use.

<table>
<thead>
<tr>
<th>Intel® MKL PARDISO Routines</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pardisoinit</td>
<td>Initializes Intel® MKL PARDISO with default parameters depending on the matrix type.</td>
</tr>
<tr>
<td>pardiso</td>
<td>Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.</td>
</tr>
<tr>
<td>Routine</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>pardiso_64</td>
<td>Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.</td>
</tr>
<tr>
<td>pardiso_getenv</td>
<td>Retrieves additional values from the Intel® MKL PARDISO handle (deprecated).</td>
</tr>
<tr>
<td>pardiso_setenv</td>
<td>Sets additional values in the Intel® MKL PARDISO handle (deprecated).</td>
</tr>
<tr>
<td>mkl_pardiso_pivot</td>
<td>Replaces routine which handles Intel® MKL PARDISO pivots with user-defined routine.</td>
</tr>
<tr>
<td>pardiso_getdiag</td>
<td>Returns diagonal elements of initial and factorized matrix.</td>
</tr>
<tr>
<td>pardiso_export</td>
<td>Places pointers dedicated for sparse representation of requested matrix into MKL PARDISO.</td>
</tr>
<tr>
<td>pardiso_handle_store</td>
<td>Store internal structures from pardiso to a file.</td>
</tr>
<tr>
<td>pardiso_handle_restore</td>
<td>Restore pardiso internal structures from a file.</td>
</tr>
<tr>
<td>pardiso_handle_delete</td>
<td>Delete files with pardiso internal structure data.</td>
</tr>
<tr>
<td>pardiso_handle_store_64</td>
<td>Store internal structures from pardiso_64 to a file.</td>
</tr>
<tr>
<td>pardiso_handle_restore_64</td>
<td>Restore pardiso_64 internal structures from a file.</td>
</tr>
<tr>
<td>pardiso_handle_delete_64</td>
<td>Delete files with pardiso_64 internal structure data.</td>
</tr>
</tbody>
</table>

**NOTE** pardiso_getenv and pardiso_setenv routines are deprecated and will be removed in Intel® MKL 2021. The routine pardiso_setenv should be replaced by either using the configuration file or by setting a corresponding environment variable MKL_PARDISO_OOC_FILE_NAME.

The Intel® MKL PARDISO solver supports a wide range of real and complex sparse matrix types (see the figure below).

---

**Sparse Matrices That Can Be Solved with the Intel MKL PARDISO Solver**

The Intel® MKL PARDISO solver performs four tasks:

- analysis and symbolic factorization
- numerical factorization
- forward and backward substitution including iterative refinement
termination to release all internal solver memory.

To find code examples that use Intel® MKL PARDISO routines to solve systems of linear equations, unzip the archive file in the examples folder of the Intel® MKL installation directory. Code examples will be in the examples/solverc/source folder.

**Supported Matrix Types**

The analysis steps performed by Intel® MKL PARDISO depend on the structure of the input matrix A.

**Symmetric Matrices**

The solver first computes a symmetric fill-in reducing permutation $P$ based on either the minimum degree algorithm [Liu85] or the nested dissection algorithm from the METIS package [Karypis98] (both included with Intel® MKL), followed by the parallel left-right looking numerical Cholesky factorization [Schenk00-2] of $PAP^T = LL^T$ for symmetric positive-definite matrices, or $PAP^T = LDL^T$ for symmetric indefinite matrices. The solver uses diagonal pivoting, or 1x1 and 2x2 Bunch-Kaufman pivoting for symmetric indefinite matrices. An approximation of $X$ is found by forward and backward substitution and optional iterative refinement.

Whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal supernode block, the coefficient matrix is perturbed. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricting notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. Furthermore, for a large set of matrices from different applications areas, this method is as accurate as a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Another method of improving the pivoting accuracy is to use symmetric weighted matching algorithms. These algorithms identify large entries in the coefficient matrix $A$ that, if permuted close to the diagonal, permit the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. These algorithms are based on maximum weighted matchings and improve the quality of the factor in a complementary way to the alternative of using more complete pivoting techniques.

The inertia is also computed for real symmetric indefinite matrices.

**Structurally Symmetric Matrices**

The solver first computes a symmetric fill-in reducing permutation $P$ followed by the parallel numerical factorization of $PAP^T = QLU^T$. The solver uses partial pivoting in the supernodes and an approximation of $X$ is found by forward and backward substitution and optional iterative refinement.

**Nonsymmetric Matrices**

The solver first computes a nonsymmetric permutation $P_{MPS}$ and scaling matrices $D_r$ and $D_c$ with the aim of placing large entries on the diagonal to enhance reliability of the numerical factorization process [Duff99]. In the next step the solver computes a fill-in reducing permutation $P$ based on the matrix $P_{MPS}A + (P_{MPS}A)^T$ followed by the parallel numerical factorization

$QLUR = PF_{MPS}D_rAD_cP$

with supernode pivoting matrices $Q$ and $R$. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99]. The magnitude of the potential pivot is tested against a constant threshold of

$alpha = eps*||A2||_inf$. 


where $\text{eps}$ is the machine precision, $A_2 = P*F_{\text{NP}}*D_r*A*D_c*P$, and $||A_2||_{\text{inf}}$ is the infinity norm of $A$. Any tiny pivots encountered during elimination are set to the sign $(1_{II})*\text{eps}||A_2||_{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Although many failures could render the factorization well-defined but essentially useless, in practice the diagonal elements are rarely modified for a large class of matrices. The result of this pivoting approach is that the factorization is, in general, not exact and iterative refinement may be needed.

**Sparse Data Storage**

Intel® MKL PARDISO stores sparse data in several formats:

- **CSR3:** The 3-array variation of the compressed sparse row format described in Three Array Variation of CSR Format.
- **BSR3:** The three-array variation of the block compressed sparse row format described in Three Array Variation of BSR Format. Use $\text{iparm}[36]$ to specify the block size.
- **VBSR:** Variable BSR format. Intel® MKL PARDISO analyzes the matrix provided in CSR3 format and converts it into an internal structure which can improve performance for matrices with a block structure. Use $\text{iparm}[36] = -t$ ($0 < t \leq 100$) to specify use of internal VBSR format and to set the degree of similarity required to combine elements of the matrix. For example, if you set $\text{iparm}[36] = -80$, two rows of the input matrix are combined when their non-zero patterns are 80% or more similar.

**NOTE**

Intel® MKL supports only the VBSR format for real and symmetric positive definite or indefinite matrices ($mtype = 2$ or $mtype = -2$).

Intel® MKL supports these features for all matrix types as long as $\text{iparm}[23] = 1$:

- $\text{iparm}[30] > 0$: Partial solution
- $\text{iparm}[35] > 0$: Schur complement
- $\text{iparm}[59] > 0$: OOC Intel® MKL PARDISO

For all storage formats, the Intel® MKL PARDISO parameter $ja$ is used for the columns array, $ia$ is used for rowIndex, and $a$ is used for values. The algorithms in Intel® MKL PARDISO require column indices $ja$ to be in increasing order per row and that the diagonal element in each row be present for any structurally symmetric matrix. For symmetric or nonsymmetric matrices the diagonal elements which are equal to zero are not necessary.

**Caution**

Intel® MKL PARDISO column indices $ja$ must be in increasing order per row. You can validate the sparse matrix structure with the matrix checker ($\text{iparm}[26]$)

**NOTE**

While the presence of zero diagonal elements for symmetric matrices is not required, you should explicitly set zero diagonal elements for symmetric matrices. Otherwise, Intel® MKL PARDISO creates internal copies of arrays $ia$, $ja$, and $a$ full of diagonal elements, which require additional memory and computational time. However, the memory and time required the diagonal elements in internal arrays is usually not significant compared to the memory and the time required to factor and solve the matrix.
Storage of Matrices

By default, Intel® MKL PARDISO stores data in RAM. This is referred to as In-Core (IC) mode. However, you can specify that Intel® MKL PARDISO store matrices on disk by setting iparm[59]. This mode is called the Out-of-Core (OOC) mode.

You can set the following parameters for the OOC mode.

<table>
<thead>
<tr>
<th>Parameter/Environment Variable Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_PARDISO_OOC_PATH</td>
<td>Directory for storing data created in the OOC mode.</td>
</tr>
<tr>
<td>MKL_PARDISO_OOC_FILE_NAME</td>
<td>Full file name (incl. path) which will be used for the OOC files</td>
</tr>
<tr>
<td>MKL_PARDISO_OOC_MAX_CORE_SIZE</td>
<td>Maximum size of RAM (in megabytes) available for Intel® MKL PARDISO</td>
</tr>
<tr>
<td>MKL_PARDISO_OOC_MAX_SWAP_SIZE</td>
<td>Maximum swap size (in megabytes) available for Intel® MKL PARDISO</td>
</tr>
<tr>
<td>MKL_PARDISO_OOC_KEEP_FILE</td>
<td>A flag which determines whether temporary data files will be deleted or stored</td>
</tr>
</tbody>
</table>

By default, the current working directory is used in the OOC mode as a directory path for storing data. All work arrays will be stored in files named ooc_temp with different extensions. When MKL_PARDISO_OOC_FILE_NAME is not set and MKL_PARDISO_OOC_PATH is set, the names for the created files will contain <path>/mkl_pardiso or <path>/mkl_pardiso depending on the OS. Setting MKL_PARDISO_OOC_FILE_NAME=<filename> will override the path which could have been set in MKL_PARDISO_OOC_PATH. In this case <filename> will be used for naming the OOC files.

By default, MKL_PARDISO_OOC_MAX_CORE_SIZE is 2000 (MB) and MKL_PARDISO_OOC_MAX_SWAP_SIZE is 0.

**NOTE**
Do not set the sum of MKL_PARDISO_OOC_MAX_CORE_SIZE and MKL_PARDISO_OOC_MAX_SWAP_SIZE greater than the size of the RAM plus the size of the swap memory. Be sure to allow enough free memory for the operating system and any other processes which need to be running.

By default, all temporary data files will be deleted. For keeping them it is required to set MKL_PARDISO_OOC_KEEP_FILE to 0.

OOC parameters can be set in a configuration file. You can set the path to this file and its name using environmental variables MKL_PARDISO_OOC_CFG_PATH and MKL_PARDISO_OOC_CFG_FILE_NAME.
For setting parameters of OOC mode either environment variables or a configuration file can be used. When the last option is chosen, by default the name of the file is pardiso_ooc.cfg and it should be placed in the working directory. If needed, the user can set the path to the configuration file using environmental variables MKL_PARDISO_OOC_CFG_PATH and MKL_PARDISO_OOC_CFG_FILE_NAME. These variables specify the path and filename as follows:

- Linux* OS and OS X*: `<MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>`
- Windows* OS: `<MKL_PARDISO_OOC_CFG_PATH>\<MKL_PARDISO_OOC_CFG_FILE_NAME>`

An example of the configuration file:

```
MKL_PARDISO_OOC_PATH = <path>
MKL_PARDISO_OOC_MAX_CORE_SIZE = N
MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

**Caution**
The maximum length of the path lines in the configuration files is 1000 characters.

Alternatively, the OOC parameters can be set as environment variables via command line.

For Linux* OS and OS X*:

```
export MKL_PARDISO_OOC_PATH = <path>
export MKL_PARDISO_OOC_MAX_CORE_SIZE = N
export MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
export MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

For Windows* OS:

```
set MKL_PARDISO_OOC_PATH = <path>
set MKL_PARDISO_OOC_MAX_CORE_SIZE = N
set MKL_PARDISO_OOC_MAX_SWAP_SIZE = K
set MKL_PARDISO_OOC_KEEP_FILE = 0 (or 1)
```

where `<path>` should follow the OS naming convention.

**Direct-Iterative Preconditioning for Nonsymmetric Linear Systems**

The solver uses a combination of direct and iterative methods [Sonn89] to accelerate the linear solution process for transient simulation. Most applications of sparse solvers require solutions of systems with gradually changing values of the nonzero coefficient matrix, but with an identical sparsity pattern. In these applications, the analysis phase of the solvers has to be performed only once and the numerical factorizations are the important time-consuming steps during the simulation. Intel® MKL PARDISO uses a numerical factorization and applies the factors in a preconditioned Krylov Subspace iteration. If the iteration does not converge, the solver automatically switches back to the numerical factorization. This method can be applied to nonsymmetric matrices in Intel® MKL PARDISO. You can select the method using the `iparm[3]` input parameter. The `iparm[19]` parameter returns the error status after running Intel® MKL PARDISO.

**Single and Double Precision Computations**

Intel® MKL PARDISO solves tasks using single or double precision. Each precision has its benefits and drawbacks. Double precision variables have more digits to store value, so the solver uses more memory for keeping data. But this mode solves matrices with better accuracy, which is especially important for input matrices with large condition numbers.

Single precision variables have fewer digits to store values, so the solver uses less memory than in the double precision mode. Additionally this mode usually takes less time. But as computations are made less precisely, only some systems of equations can be solved accurately enough using single precision.
Separate Forward and Backward Substitution

The solver execution step (see parameter $phase = 33$ below) can be divided into two or three separate substitutions: forward, backward, and possible diagonal. This separation can be explained by the examples of solving systems with different matrix types.

A real symmetric positive definite matrix $A$ ($mtype = 2$) is factored by Intel® MKL PARDISO as $A = L^*L^T$. In this case the solution of the system $A^*x=b$ can be found as sequence of substitutions: $L^*y=b$ (forward substitution, $phase = 331$) and $L^T*x=y$ (backward substitution, $phase = 333$).

A real nonsymmetric matrix $A$ ($mtype = 11$) is factored by Intel® MKL PARDISO as $A = L^*U$. In this case the solution of the system $A^*x=b$ can be found by the following sequence: $L^*y=b$ (forward substitution, $phase = 331$) and $U^*x=y$ (backward substitution, $phase = 333$).

Solving a system with a real symmetric indefinite matrix $A$ ($mtype = -2$) is slightly different from the cases above. Intel® MKL PARDISO factors this matrix as $A = LDL^T$, and the solution of the system $A^*x=b$ can be calculated as the following sequence of substitutions: $L^*y=b$ (forward substitution, $phase = 331$), $D^*v=y$ (diagonal substitution, $phase = 332$), and finally $L^T*x=v$ (backward substitution, $phase = 333$). Diagonal substitution makes sense only for symmetric indefinite matrices ($mtype = -2, -4, 6$). For matrices of other types a solution can be found as described in the first two examples.

Caution

The number of refinement steps ($iparm[7]$) must be set to zero if a solution is calculated with separate substitutions ($phase = 331, 332, 333$), otherwise Intel® MKL PARDISO produces the wrong result.

NOTE

Different pivoting ($iparm[20]$) produces different $LDL^T$ factorization. Therefore results of forward, diagonal and backward substitutions with diagonal pivoting can differ from results of the same steps with Bunch-Kaufman pivoting. Of course, the final results of sequential execution of forward, diagonal and backward substitution are equal to the results of the full solving step ($phase = 33$) regardless of the pivoting used.

Callback Function for Pivoting Control

In-core Intel® MKL PARDISO allows you to control pivoting with a callback routine, mkl_pardiso_pivot. You can then use the pardiso_getdiag routine to access the diagonal elements. Set $iparm[55]$ to 1 in order to use the callback functionality.

Low Rank Update

Use low rank update to accelerate the factorization step in Intel® MKL PARDISO when you use multiple matrices with identical structure and similar values. After calling pardiso in the usual manner for factorization ($phase = 12, 13, 22, or 23$) for some matrix $A1$, low rank update can be applied to the factorization step ($phase = 22 or 23$) of some matrix $A2$ with identical structure.

To use the low rank update feature, set $iparm[38] = 1$ while also setting $iparm[23] = 10$. Additionally, supply an array that lists the values in $A2$ that are different from $A1$ using the $perm$ parameter as outlined in the pardiso $perm$ parameter description.
Important
Low rank update can only be called for matrices with the exact same pattern of nonzero values. As such, the value of the mtype, ia, ja, and iparm[23] parameters should also be identical. In general, the low rank factorization should be called with the same parameters as the preceding factorization step for the same internal data structure handle (except for array a, iparm[38], and perm).

Low rank update does not currently support Intel TBB threading. In this case, Intel® MKL PARDISO defaults to full factorization instead.

Low rank update cannot be used in combination with a user-supplied permutation vector - in other words, you must use the default values of iparm[4] = 0, iparm[30] = 0, and iparm[35] = 0. Additionally, iparm[3], iparm[5], iparm[27], iparm[36], iparm[55], and iparm[59] must all be set to the default value of 0.

pardiso
Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.

Syntax
void pardiso (_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT *mnum, const MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n, const void *a, const MKL_INT *ia, const MKL_INT *ja, MKL_INT *perm, const MKL_INT *nrhs, MKL_INT *iparm, const MKL_INT *msglvl, void *b, void *x, MKL_INT *error);

Include Files
• mkl.h

Description
The routine pardiso calculates the solution of a set of sparse linear equations

$$A \times X = B$$

with single or multiple right-hand sides, using a parallel LU, LDL, or LLᵀ factorization, where A is an n-by-n matrix, and X and B are n-by-nrhs vectors or matrices.

NOTE
This routine supports the Progress Routine feature. See Progress Function section for details. The case of iparm[23]=10 does not support this feature.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Input Parameters

pt

Array with size of 64.
Handle to internal data structure. The entries must be set to zero prior to the first call to pardiso. Unique for factorization.

**Caution**
After the first call to pardiso do not directly modify pt, as that could cause a serious memory leak.

Use the pardiso_handle_store or pardiso_handle_store_64 routine to store the content of pt to a file. Restore the contents of pt from the file using pardiso_handle_restore or pardiso_handle_restore_64. Use pardiso_handle_store and pardiso_handle_restore with pardiso, and pardiso_handle_store_64 and pardiso_handle_restore_64 with pardiso_64.

maxfct

Maximum number of factors with identical sparsity structure that must be kept in memory at the same time. In most applications this value is equal to 1. It is possible to store several different factorizations with the same nonzero structure at the same time in the internal data structure management of the solver.

pardiso can process several matrices with an identical matrix sparsity pattern and it can store the factors of these matrices at the same time. Matrices with a different sparsity structure can be kept in memory with different memory address pointers pt.

mnum

Indicates the actual matrix for the solution phase. With this scalar you can define which matrix to factorize. The value must be: 1 \leq mnum \leq maxfct.
In most applications this value is 1.

mtype

Defines the matrix type, which influences the pivoting method. The Intel® MKL PARDISO solver supports the following matrices:

1    real and structurally symmetric
2    real and symmetric positive definite
-2   real and symmetric indefinite
3    complex and structurally symmetric
4    complex and Hermitian positive definite
-4   complex and Hermitian indefinite
6    complex and symmetric
11   real and nonsymmetric
13   complex and nonsymmetric
Controls the execution of the solver. Usually it is a two- or three-digit integer. The first digit indicates the starting phase of execution and the second digit indicates the ending phase. Intel® MKL PARDISO has the following phases of execution:

- Phase 1: Fill-reduction analysis and symbolic factorization
- Phase 2: Numerical factorization
- Phase 3: Forward and Backward solve including optional iterative refinement

This phase can be divided into two or three separate substitutions: forward, backward, and diagonal (see Separate Forward and Backward Substitution).

- Memory release phase (phase = 0 or phase = -1)

If a previous call to the routine has computed information from previous phases, execution may start at any phase. The phase parameter can have the following values:

<table>
<thead>
<tr>
<th>phase</th>
<th>Solver Execution Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Analysis</td>
</tr>
<tr>
<td>12</td>
<td>Analysis, numerical factorization</td>
</tr>
<tr>
<td>13</td>
<td>Analysis, numerical factorization, solve, iterative refinement</td>
</tr>
<tr>
<td>22</td>
<td>Numerical factorization</td>
</tr>
<tr>
<td>23</td>
<td>Numerical factorization, solve, iterative refinement</td>
</tr>
<tr>
<td>33</td>
<td>Solve, iterative refinement</td>
</tr>
<tr>
<td>331</td>
<td>like phase=33, but only forward substitution</td>
</tr>
<tr>
<td>332</td>
<td>like phase=33, but only diagonal substitution (if available)</td>
</tr>
<tr>
<td>333</td>
<td>like phase=33, but only backward substitution</td>
</tr>
<tr>
<td>0</td>
<td>Release internal memory for L and U matrix number mnum</td>
</tr>
<tr>
<td>-1</td>
<td>Release all internal memory for all matrices</td>
</tr>
</tbody>
</table>

If iparm[35] = 0, phases 331, 332, and 333 perform this decomposition:

\[
A = \begin{bmatrix}
L_{11} & 0 & D_{11} & 0 & U_{11} & U_{21} \\
L_{12} & L_{22} & 0 & D_{22} & 0 & U_{22}
\end{bmatrix}
\]

If iparm[35] = 2, phases 331, 332, and 333 perform a different decomposition:

\[
A = \begin{bmatrix}
L_{11} & 0 & I & 0 & U_{11} & U_{21} \\
L_{12} & I & 0 & S & 0 & I
\end{bmatrix}
\]
You can supply a custom implementation for phase 332 instead of calling 
pardiso. For example, it can be implemented with dense LAPACK 
functionality. Custom implementation also allows you to substitute the 
matrix $S$ with your own.

**NOTE**
For very large Schur complement matrices use LAPACK 
functionality to compute the Schur complement vector instead 
of the Intel® MKL PARDISO phase 332 implementation.

**n**
Number of equations in the sparse linear systems of equations $A*X = B$. 
Constraint: $n > 0$.

**a**
Array. Contains the non-zero elements of the coefficient matrix $A$ 
corresponding to the indices in $ja$. The coefficient matrix can be either real 
or complex. The matrix must be stored in the three-array variant of the 
compressed sparse row (CSR3) or in the three-array variant of the block 
compressed sparse row (BSR3) format, and the matrix must be stored with 
increasing values of $ja$ for each row.

For CSR3 format, the size of $a$ is the same as that of $ja$. Refer to the 
values array description in Three Array Variation of CSR Format for more 
details.

For BSR3 format the size of $a$ is the size of $ja$ multiplied by the square of 
the block size. Refer to the values array description in Three Array 
Variation of BSR Format for more details.

**NOTE**
If you set $iparm[36]$ to a negative value, Intel® MKL PARDISO 
converts the data from CSR3 format to an internal variable BSR 
(VBSR) format. See Sparse Data Storage.

**ia**
Array, size $(n+1)$.

For CSR3 format, $ia[i] \ (i<n)$ points to the first column index of row $i$ in 
the array $ja$. That is, $ia[i]$ gives the index of the element in array $a$ that 
contains the first non-zero element from row $i$ of $A$. The last element $ia[n]$ 
is taken to be equal to the number of non-zero elements in $A$, plus one. 
Refer to $rowIndex$ array description in Three Array Variation of CSR Format 
for more details.

For BSR3 format, $ia[i] \ (i<n)$ points to the first column index of row $i$ in 
the array $ja$. That is, $ia[i]$ gives the index of the element in array $a$ that 
contains the first non-zero block from row $i$ of $A$. The last element $ia[n]$ is 
taken to be equal to the number of non-zero blocks in $A$, plus one. Refer to 
$rowIndex$ array description in Three Array Variation of BSR Format for more 
details.

The array $ia$ is accessed in all phases of the solution process.

Indexing of $ia$ is one-based by default, but it can be changed to zero-based 
by setting the appropriate value to the parameter $iparm[34]$. 
For CSR3 format, array \( ja \) contains column indices of the sparse matrix \( A \).
It is important that the indices are in increasing order per row. For
structurally symmetric matrices it is assumed that all diagonal elements are
stored (even if they are zeros) in the list of non-zero elements in \( a \) and \( ja \).
For symmetric matrices, the solver needs only the upper triangular part of
the system as is shown for \( columns \) array in Three Array Variation of CSR
Format.

For BSR3 format, array \( ja \) contains column indices of the sparse matrix \( A \).
It is important that the indices are in increasing order per row. For
structurally symmetric matrices it is assumed that all diagonal blocks are
stored (even if they are zeros) in the list of non-zero blocks in \( a \) and \( ja \).
For symmetric matrices, the solver needs only the upper triangular part of
the system as is shown for \( columns \) array in Three Array Variation of BSR
Format.

The array \( ja \) is accessed in all phases of the solution process.

Indexing of \( ja \) is one-based by default, but it can be changed to zero-based
by setting the appropriate value to the parameter \( iparm[34] \).

Array, size \((n)\). Depending on the value of \( iparm[4] \) and \( iparm[30] \), holds
the permutation vector of size \( n \), specifies elements used for computing a
partial solution, or specifies differing values of the input matrices for low
rank update.

- If \( iparm[4] = 1, iparm[30] = 0, \) and \( iparm[35] = 0 \), \( perm \) specifies
  the fill-in reducing ordering to the solver. Let \( A \) be the original matrix
  and \( C = P^*A^*P^T \) be the permuted matrix. Row (column) \( i \) of \( C \) is the
  \( perm[i] \) row (column) of \( A \). The array \( perm \) is also used to return the
  permutation vector calculated during fill-in reducing ordering stage.

  **NOTE**
  Be aware that setting \( iparm[4] = 1 \) prevents use of a parallel
  algorithm for the solve step.

- If \( iparm[4] = 2, iparm[30] = 0, \) and \( iparm[35] = 0 \), the permutation
  vector computed in phase 11 is returned in the \( perm \) array.
- If \( iparm[4] = 0, iparm[30] > 0, \) and \( iparm[35] = 0 \), \( perm \) specifies
  elements of the right-hand side to use or of the solution to compute for
  a partial solution.
- If \( iparm[4] = 0, iparm[30] = 0, \) and \( iparm[35] > 0 \), \( perm \) specifies
  elements for a Schur complement.
- If \( iparm[38] = 1, \) \( perm \) specifies values that differ in \( A \) for low rank
  update (see Low Rank Update). The size of the array must be at least
  \( 2*ndiff + 1 \), where \( ndiff \) is the number of values of \( A \) that are different.
  The values of \( perm \) should be:
  \[
  perm = \{ ndiff, row_index1, column_index1, row_index2, column_index2, ..., row_index_ndiff, column_index_ndiff \}
  \]
  where \( row_index_m \) and \( column_index_m \) are the row and column
  indices of the \( m \)-th differing non-zero value in matrix \( A \). The row and
  column index pairs can be in any order, but must use zero-based
  indexing regardless of the value of \( iparm[34] \).
See iparm[4], iparm[30], and iparm[38] for more details.

Indexing of perm is one-based by default, but unless iparm[38] = 1 it can be changed to zero-based by setting the appropriate value to the parameter iparm[34].

**nrhs**
Number of right-hand sides that need to be solved for.

**iparm**
Array, size (64). This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver.

See pardiso iparm Parameter for more details about the iparm parameters.

**msglvl**
Message level information. If msglvl = 0 then pardiso generates no output, if msglvl = 1 the solver prints statistical information to the screen.

**b**
Array, size (n*nrhs). On entry, contains the right-hand side vector/matrix B, which is placed in memory contiguously. The b[k*nrhs] element must hold the i-th component of k-th right-hand side vector. Note that b is only accessed in the solution phase.

### Output Parameters

(See also Intel MKL PARDISO Parameters in Tabular Form.)

**pt**
Handle to internal data structure.

See the Input Parameter description of the perm array.

**perm**
On output, some iparm values report information such as the numbers of non-zero elements in the factors.

See pardiso iparm Parameter for more details about the iparm parameters.

**iparm**
On output, the array is replaced with the solution if iparm[5] = 1.

**b**
Array, size (n*nrhs). If iparm[5]=0 it contains solution vector/matrix X, which is placed contiguously in memory. The x[i + k*n] element must hold the i-th component of the k-th solution vector. Note that x is only accessed in the solution phase.

**error**
The error indicator according to the below table:

<table>
<thead>
<tr>
<th>error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>input inconsistent</td>
</tr>
<tr>
<td>-2</td>
<td>not enough memory</td>
</tr>
<tr>
<td>-3</td>
<td>reordering problem</td>
</tr>
<tr>
<td>-4</td>
<td>Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (iparm[9]) and also increase the number of iterative refreshments.</td>
</tr>
<tr>
<td>error</td>
<td>Information</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>-5</td>
<td>unclassified (internal) error</td>
</tr>
<tr>
<td>-6</td>
<td>reordering failed (matrix types 11 and 13 only)</td>
</tr>
<tr>
<td>-7</td>
<td>diagonal matrix is singular</td>
</tr>
<tr>
<td>-8</td>
<td>32-bit integer overflow problem</td>
</tr>
<tr>
<td>-9</td>
<td>not enough memory for OOC</td>
</tr>
<tr>
<td>-10</td>
<td>error opening OOC files</td>
</tr>
<tr>
<td>-11</td>
<td>read/write error with OOC files</td>
</tr>
<tr>
<td>-12</td>
<td>(pardiso_64 only) pardiso_64 called from 32-bit library</td>
</tr>
<tr>
<td>-13</td>
<td>interrupted by the (user-defined) mkl_progress function</td>
</tr>
</tbody>
</table>

**pardisoinit**

Initialize Intel® MKL PARDISO with default parameters in accordance with the matrix type.

**Syntax**

```c
void pardisoinit (_MKL_DSS_HANDLE_t pt, const MKL_INT *mtype, MKL_INT *iparm );
```

**Include Files**

- mkl.h

**Description**

This function initializes Intel® MKL PARDISO internal address pointer `pt` with zero values (as needed for the very first call of pardiso) and sets default `iparm` values in accordance with the matrix type. Intel® MKL supplies the pardisoinit routine to be compatible with PARDISO 3.2 or lower.

**NOTE**

An alternative way to set default `iparm` values is to call pardiso in the analysis phase with `iparm(1)=0`. In this case you must initialize the internal address pointer `pt` with zero values manually.
NOTE
The pardisoinit routine initializes only the in-core version of Intel® MKL PARDISO. Switching on the out-of-core version of Intel® MKL PARDISO as well as changing default iparm values can be done after the call to pardisoinit but before the first call to pardiso.

Input Parameters

perm
Ignored.

mtype
This scalar value defines the matrix type. Based on this value pardisoinit sets default values for the iparm array. Refer to the section Intel MKL PARDISO Parameters in Tabular Form for more details about the default values of Intel® MKL PARDISO.

Output Parameters

pt
Array with a dimension of 64. Solver internal data address pointer. These addresses are passed to the solver, and all related internal memory management is organized through this array. The pardisoinit routine nullifies the array pt.

NOTE
It is very important that the pointer pt is initialized with zero before the first call of Intel® MKL PARDISO. After that first call you should never modify the pointer, because it could cause a serious memory leak or a crash.

iparm
Array with a dimension of 64. This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver. The pardisoinit routine fills-in the iparm array with the default values. Refer to the section Intel MKL PARDISO Parameters in Tabular Form for more details about the default values of Intel® MKL PARDISO.

pardiso_64
Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides, 64-bit integer version.

Syntax

void pardiso_64 (_MKL_DSS_HANDLE_t pt, const long long int *maxfct, const long long int *mnum, const long long int *mtype, const long long int *phase, const long long int *n, const void *a, const long long int *ia, const long long int *ja, long long int *perm, const long long int *nrhs, long long int *iparm, const long long int *msglvl, void *b, void *x, long long int *error);

Include Files

• mkl.h
Description

pardiso_64 is an alternative ILP64 (64-bit integer) version of the pardiso routine (see Description section for more details). The interface of pardiso_64 is the same as the interface of pardiso, but it accepts and returns all integer data as long long int.

Use pardiso_64 when pardiso for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® MKL functionality. In other words, if you use 64-bit integer version (pardiso_64), you do not need to re-link your applications with ILP64 libraries. Take into account that pardiso_64 may perform slower than regular pardiso on the reordering and symbolic factorization phase.

NOTE
pardiso_64 is supported only in the 64-bit libraries. If pardiso_64 is called from the 32-bit libraries, it returns error =-12.

NOTE
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

The input parameters of pardiso_64 are the same as the input parameters of pardiso, but pardiso_64 accepts all integer data as long long int.

Output Parameters

The output parameters of pardiso_64 are the same as the output parameters of pardiso, but pardiso_64 returns all integer data as long long int.

pardiso_getenv, pardiso_setenv

Retrieves additional values from or sets additional values in the Intel® MKL PARDISO handle (deprecated).

Syntax

MKL_INT pardiso_getenv (const _MKL_DSS_HANDLE_t handle, const enum PARDISO_ENV_PARAM *param, char *value);
MKL_INT pardiso_setenv (_MKL_DSS_HANDLE_t handle, const enum PARDISO_ENV_PARAM *param, const char *value);

Include Files

• mkl.h

Description

These functions operate with the Intel® MKL PARDISO handle. The pardiso_getenv routine retrieves additional values from the Intel® MKL PARDISO handle, and pardiso_setenv sets specified values in the Intel® MKL PARDISO handle.

These functions enable retrieving and setting the name of the Intel® MKL PARDISO OOC file.
To retrieve the Intel® MKL PARDISO OOC file name, you can apply this function to any properly-created handle.

To set the Intel® MKL PARDISO OOC file name in the handle you must call the function before the reordering stage. This is because the OOC file name is stored in the handle after the reordering stage and it is not changed during further computations.

NOTE
A 1024-byte internal buffer is used inside Intel® MKL PARDISO for storing the OOC file name. Allocate a 1024-byte buffer for passing to the `pardiso_getenv` function as the value parameter.

**Input Parameters**

**handle**

Intel® MKL PARDISO handle for which to set and from which to retrieve information. (See DSS Interface Description for structure description)

**param**

Specifies the required parameter. The only value is `PARDISO_OOC_FILE_NAME`, defined in the corresponding include file.

**value**

Input parameter for `pardiso_setenv`. Contains the name of the OOC file that must be used in the handle.

**Output Parameters**

**handle**

Output parameter for `pardiso_setenv`. Data object of the MKL_DSS_HANDLE type (see DSS Interface Description).

**value**

Output parameter for `pardiso_getenv`. Contains the name of the OOC file which must be used in the handle.

**mkl_pardiso_pivot**

Replaces routine which handles Intel® MKL PARDISO pivots with user-defined routine.

**Syntax**

```c
void mkl_pardiso_pivot (const void *ai, void *bi, const void *eps);
```

**Include Files**

- mkl.h

**Description**

The `mkl_pardiso_pivot` routine allows you to handle diagonal elements which arise during numerical factorization that are zero or near zero. By default, Intel® MKL PARDISO determines that a diagonal element $bi$ is a pivot if $bi < eps$, and if so, replaces it with $eps$. But you can provide your own routine to modify the resulting factorized matrix in case there are small elements on the diagonal during the factorization step.

NOTE
To use this routine, you must set `iparm[55]` to 1 before the main `pardiso` loop.
Input Parameters

- $ai$: Diagonal element of initial matrix corresponding to pivot element.
- $bi$: Diagonal element of factorized matrix that could be chosen as a pivot element.
- $\epsilon$: Scalar to compare with diagonal of factorized matrix. On input equal to parameter described by $iparm[9]$.

Output Parameters

- $bi$: In case element is chosen as a pivot, value with which to replace the pivot.

**pardiso_getdiag**

_Returns diagonal elements of initial and factorized matrix._

**Syntax**

```c
void pardiso_getdiag (const _MKL_DSS_HANDLE_t pt, void *df, void *da, const MKL_INT *mnum, MKL_INT *error);
```

**Include Files**

- mkl.h

**Description**

This routine returns the diagonal elements of the initial and factorized matrix for a real or Hermitian matrix.

**NOTE**

In order to use this routine, you must set $iparm[55]$ to 1 before the main _pardiso_ loop.

**Input Parameters**

- $pt$: Array with a size of 64. Handle to internal data structure for the Intel® MKL PARDISO solver. The entries must be set to zero prior to the first call to _pardiso_. Unique for factorization.

- $mnum$: Indicates the actual matrix for the solution phase of the Intel® MKL PARDISO solver. With this scalar you can define the diagonal elements of the factorized matrix that you want to obtain. The value must be: $1 \leq mnum \leq maxfct$. In most applications this value is 1.

**Output Parameters**

- $df$: Array with a dimension of $n$. Contains diagonal elements of the factorized matrix after factorization.
**NOTE**
Elements of $df$ correspond to diagonal elements of matrix $L$ computed during phase 22. Because during phase 22 Intel® MKL PARDISO makes additional permutations to improve stability, it is possible that array $df$ is not in line with the $perm$ array computed during phase 11.

$da$
Array with a dimension of $n$. Contains diagonal elements of the initial matrix.

**NOTE**
Elements of $da$ correspond to diagonal elements of matrix $L$ computed during phase 22. Because during phase 22 Intel® MKL PARDISO makes additional permutations to improve stability, it is possible that array $da$ is not in line with the $perm$ array computed during phase 11.

$error$
The error indicator.

<table>
<thead>
<tr>
<th>error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>Diagonal information not turned on before pardiso main loop ($iparm[55]=0$).</td>
</tr>
</tbody>
</table>

**pardiso_export**
Places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO

**Syntax**
```c
void pardiso_export (const _MKL_DSS_HANDLE_t pt, void* values, MKL_INT* rows, MKL_INT* columns, MKL_INT* step, MKL_INT* iparm, MKL_INT* error);
```

**Include Files**
- mkl.h

**Description**
This auxiliary routine places pointers dedicated for sparse representation of a requested matrix (values, rows, and columns) into MKL PARDISO. The matrix will be stored in the three-array variant of the compressed sparse row (CSR3 format) with 0-based indexing. For details, refer to Three Array Variation of CSR Format.

**NOTE**
Currently, this routine can be used only for a sparse Schur complement matrix. All parameters related to the Schur complement matrix ($perm$, $iparm$) must be set before the reordering stage of MKL PARDISO (phase = 11) is called.
**Input Parameters**

*pt*

Array with a size of 64. Handle to internal data structure for the Intel® MKL PARDISO solver. The entries must be set to zero prior to the first call to `pardiso`. Unique for factorization.

*iparm*

This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver.

*step*

Stage indicator. These are the currently supported values:

<table>
<thead>
<tr>
<th>Step value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Used to place pointers related to a Schur complement matrix in MKL PARDISO. The routine with <code>step</code> equal to 1 must be called between the reordering and factorization phases of MKL PARDISO.</td>
</tr>
<tr>
<td>-1</td>
<td>Used to clean the internal handle.</td>
</tr>
</tbody>
</table>

**Input/Output Parameters**

*values*

Parameter type: input/output parameter.

This array contains the non-zero elements of the requested matrix.

*rows*

Parameter type: input/output parameter.

Array of size (`size` + 1)

For CSR3 format, `rows[i]` (i < `size`) points to the first column index of row `i` in the array `columns`; that is, `rows[i]` gives the index of the element in the array `values` that contains the first non-zero element from row `i` of the sparse matrix. The last element, `rows[size]`, is equal to the number of non-zero elements in the sparse matrix.

*columns*

Parameter type: input/output parameter.

This array contains the column indices for the non-zero elements of the requested matrix.

*error*

Parameter type: output parameter.

The error status:

- 0 indicates no error.
- 1 indicates inconsistent input data.

**Usage Example**

The following C-style example demonstrates how to use the `pardiso_export` routine to get the sparse representation (that is, three-array CSR format) of a Schur complement matrix.

```c
#include "mkl.h"

/*
 * Call the reordering phase of MKL PARDISO with iparm[35] set to -1 in
 * order to compute the Schur complement matrix only, or -2 to compute all
```
* factorization arrays. perm array indices related to the Schur complement
  * matrix must be set to 1.
  */
phase = 11;
for ( i = 0; i < schur_size; i++ ) { perm[i] = 1.; }
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
    iparm, &msglvl, b, x, &error);

/*
 * After the reordering phase, iparm[35] will contain the number of non-zero
 * elements for the Schur complement matrix. Arrays dedicated to the sparse
 * representation of the Schur complement matrix must be allocated before
 * the factorization stage of MKL PARDISO is called.
 */
schur_nnz     = iparm[35];
schur_rows    = (MKL_INT   *) mkl_malloc(schur_size+1, ALIGNMENT);
schur_columns = (MKL_INT   *) mkl_malloc(schur_nnz , ALIGNMENT);
schur_values  = (DATA_TYPE *) mkl_malloc(schur_nnz , ALIGNMENT);

/*
 * Call to the pardiso_export routine with step equal to 1 in order to put
 * pointers related to the three-array CSR format into MKL PARDISO:
 */
pardiso_export(pt, schur_values, schur_ia, schur_ia, &step, iparm, &error);

/*
 * Call the factorization phase of PARDISO with iparm[35] equal to -1 or -2
 * to compute the Schur complement matrix:
 */
phase = 22;
iparm[35] = -1;
pardiso(pt, &maxfct, &mnum, &mtype, &phase, &n, a, ia, ja, perm, &nrhs,
    iparm, &msglvl, b, x, &error);

/*
 * After the factorization stage, schur_values, schur_rows, and
 * schur_columns will contain the Schur complement matrix in CSR3 format.
 */

**pardiso_handle_store**

*Store internal structures from `pardiso` to a file.*

**Syntax**

```c
void pardiso_handle_store (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

**Include Files**

- mkl.h

**Description**

This function stores Intel® MKL PARDISO structures to a file, allowing you to store Intel® MKL PARDISO internal structures between the stages of the `pardiso` routine. The `pardiso_handle_restorer` routine can restore the Intel® MKL PARDISO internal structures from the file.
Input Parameters

*pt*  
Array with a size of 64. Handle to internal data structure.

*dirname*  
String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string (""") to specify the current directory. The routine creates a file named handle.pds in the directory.

Output Parameters

*pt*  
Handle to internal data structure.

*error*  
The error indicator.

<table>
<thead>
<tr>
<th>error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-2</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot open file for writing.</td>
</tr>
<tr>
<td>-11</td>
<td>Error while writing to file.</td>
</tr>
<tr>
<td>-13</td>
<td>Wrong file format.</td>
</tr>
</tbody>
</table>

**pardiso_handle_restore**

*Restore pardiso internal structures from a file.*

**Syntax**

```c
void pardiso_handle_restore (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

**Include Files**

- mkl.h

**Description**

This function restores Intel® MKL PARDISO structures from a file. This allows you to restore Intel® MKL PARDISO internal structures stored by `pardiso_handle_store` after a phase of the `pardiso` routine and continue execution of the next phase.

Input Parameters

*dirname*  
String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string (""") to specify the current directory.

Output Parameters

*pt*  
Array with a dimension of 64. Handle to internal data structure.

*error*  
The error indicator.
<table>
<thead>
<tr>
<th><strong>error</strong></th>
<th><strong>Information</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-2</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot open file for reading.</td>
</tr>
<tr>
<td>-11</td>
<td>Error while reading from file.</td>
</tr>
<tr>
<td>-13</td>
<td>Wrong file format.</td>
</tr>
</tbody>
</table>

**pardiso_handle_delete**

Delete files with pardiso internal structure data.

**Syntax**

```c
void pardiso_handle_delete (const char *dirname, MKL_INT *error);
```

**Include Files**

- mkl.h

**Description**

This function deletes files generated with pardiso_handle_store that contain Intel® MKL PARDISO internal structures.

**Input Parameters**

- **dirname**
  
  String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

**Output Parameters**

- **error**
  
  The error indicator.

<table>
<thead>
<tr>
<th><strong>error</strong></th>
<th><strong>Information</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot delete files.</td>
</tr>
</tbody>
</table>

**pardiso_handle_store_64**

Store internal structures from pardiso_64 to a file.

**Syntax**

```c
void pardiso_handle_store_64 (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

**Include Files**

- mkl.h
**Description**

This function stores Intel® MKL PARDISO structures to a file, allowing you to store Intel® MKL PARDISO internal structures between the stages of the `pardiso_64` routine. The `pardiso_handle_restore_64` routine can restore the Intel® MKL PARDISO internal structures from the file.

**Input Parameters**

- `pt`: Array with a dimension of 64. Handle to internal data structure.
- `dirname`: String containing the name of the directory to which to write the files with the content of the internal structures. Use an empty string (""") to specify the current directory. The routine creates a file named `handle.pds` in the directory.

**Output Parameters**

- `pt`: Handle to internal data structure.
- `error`: The error indicator.

<table>
<thead>
<tr>
<th><code>error</code></th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-2</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot open file for writing.</td>
</tr>
<tr>
<td>-11</td>
<td>Error while writing to file.</td>
</tr>
<tr>
<td>-12</td>
<td>Not supported in 32-bit library - routine is only supported in 64-bit libraries.</td>
</tr>
<tr>
<td>-13</td>
<td>Wrong file format.</td>
</tr>
</tbody>
</table>

**pardiso_handle_restore_64**

*Restore pardiso_64 internal structures from a file.*

**Syntax**

```c
void pardiso_handle_restore_64 (_MKL_DSS_HANDLE_t pt, const char *dirname, MKL_INT *error);
```

**Include Files**

- `mkl.h`

**Description**

This function restores Intel® MKL PARDISO structures from a file. This allows you to restore Intel® MKL PARDISO internal structures stored by `pardiso_handle_store_64` after a phase of the `pardiso_64` routine and continue execution of the next phase.
**Input Parameters**

*dirname*

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

**Input Parameters**

*pt*

Array with a dimension of 64. Handle to internal data structure.

*error*

The error indicator.

<table>
<thead>
<tr>
<th><strong>error</strong></th>
<th><strong>Information</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-2</td>
<td>Not enough memory.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot open file for reading.</td>
</tr>
<tr>
<td>-11</td>
<td>Error while reading from file.</td>
</tr>
<tr>
<td>-13</td>
<td>Wrong file format.</td>
</tr>
</tbody>
</table>

**pardiso_handle_delete_64**

**Syntax**

Delete files with pardiso_64 internal structure data.

```c
void pardiso_handle_delete_64 (const char *dir, MKL_INT *error);
```

**Include Files**

- mkl.h

**Description**

This function deletes files generated with pardiso_handle_store_64 that contain Intel® MKL PARDISO internal structures.

**Input Parameters**

*dirname*

String containing the name of the directory in which the file with the content of the internal structures are located. Use an empty string ("") to specify the current directory.

**Output Parameters**

*error*

The error indicator.

<table>
<thead>
<tr>
<th><strong>error</strong></th>
<th><strong>Information</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No error.</td>
</tr>
<tr>
<td>-10</td>
<td>Cannot delete files.</td>
</tr>
<tr>
<td>-12</td>
<td>Not supported in 32-bit library - routine is only supported in 64-bit libraries.</td>
</tr>
</tbody>
</table>
**Intel® MKL PARDISO Parameters in Tabular Form**

The following table lists all parameters of Intel® MKL PARDISO and gives their brief descriptions.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Values</th>
<th>Comments</th>
<th>In/Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>pt</td>
<td>void*</td>
<td>Solver internal data address pointer</td>
<td>0</td>
<td>Must be initialized with zeros and never be modified later</td>
<td>in/out</td>
</tr>
<tr>
<td>maxfct</td>
<td>MKL_INT*</td>
<td>Maximal number of factors in memory</td>
<td>&gt;0</td>
<td>Generally used value is 1</td>
<td>in</td>
</tr>
<tr>
<td>nnum</td>
<td>MKL_INT*</td>
<td>The number of matrix (from 1 to maxfct) to solve</td>
<td>[1: maxfct]</td>
<td>Generally used value is 1</td>
<td>in</td>
</tr>
<tr>
<td>mtype</td>
<td>MKL_INT*</td>
<td>Matrix type</td>
<td>1</td>
<td>Real and structurally symmetric</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>Real and symmetric positive definite</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-2</td>
<td>Real and symmetric indefinite</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>Complex and structurally symmetric</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>Complex and Hermitian positive definite</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-4</td>
<td>Complex and Hermitian indefinite</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>Complex and symmetric matrix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>Real and nonsymmetric matrix</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>Complex and nonsymmetric matrix</td>
<td></td>
</tr>
<tr>
<td>phase</td>
<td>MKL_INT*</td>
<td>Controls the execution of the solver</td>
<td>11</td>
<td>Analysis</td>
<td>in</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>Analysis, numerical factorization</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>13</td>
<td>Analysis, numerical factorization, solve</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td>Numerical factorization</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23</td>
<td>Numerical factorization, solve</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td>Solve, iterative refinement</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>331</td>
<td>phase=33, but only forward substitution</td>
<td></td>
</tr>
</tbody>
</table>

*Sparse Solver Routines*
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>MKL_INT</td>
<td>Number of equations in the sparse linear system $A\times X = B$</td>
<td>&gt;0</td>
<td>in</td>
</tr>
<tr>
<td>a</td>
<td>void*</td>
<td>Contains the non-zero elements of the coefficient matrix $A$</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td>ia[n ]</td>
<td>MKL_INT</td>
<td>rowIndex array in CSR3 format</td>
<td>&gt;=0</td>
<td>in</td>
</tr>
<tr>
<td>ja</td>
<td>MKL_INT</td>
<td>columns array in CSR3 format</td>
<td>&gt;=0</td>
<td>in</td>
</tr>
</tbody>
</table>

**Values**

- **332** $\text{phase}=33$, but only diagonal substitution
- **333** $\text{phase}=33$, but only backward substitution
- **0** Release internal memory for L and U of the matrix number $mnum$
- **-1** Release all internal memory for all matrices

**Comments**

- The size of $a$ is the same as that of $ja$, and the coefficient matrix can be either real or complex. The matrix must be stored in the 3-array variation of compressed sparse row (CSR3) format with increasing values of $ja$ for each row.

- $ia[i]$ gives the index of the element in array $a$ that contains the first non-zero element from row $i$ of $A$. The last element $ia(n)$ is taken to be equal to the number of non-zero elements in $A$.

- **Note:** $iparm[34]$ indicates whether row/column indexing starts from 1 or 0.

- The column indices for each row of $A$ must be sorted in increasing order. For structurally symmetric matrices zero diagonal elements must be stored in $a$ and $ja$. Zero diagonal elements should be stored for symmetric matrices, although they are not required. For symmetric matrices, the solver needs only the upper triangular part of the system.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Values</th>
<th>Comments</th>
<th>In/Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>perm[n]</td>
<td>MKL_INT*</td>
<td>Holds the permutation vector of size $n$, specifies elements used for computing a partial solution, or specifies differing values of the input matrices for low rank update.</td>
<td>$\geq 0$</td>
<td>Note: $iparm[34]$ indicates whether row/column indexing starts from 1 or 0. You can apply your own fill-in reducing ordering ($iparm[4]=1$) or return the permutation from the solver ($iparm[4]=2$). Let $C = P^t A^t P$ be the permuted matrix. Row (column) $i$ of $C$ is the $perm(i)$ row (column) of $A$. The numbering of the array must describe a permutation. To specify elements for a partial solution, set $iparm[4]=0$, $iparm[30]&gt;0$, and $iparm[35]=0$. To specify elements for a Schur complement, set $iparm[4]=0$, $iparm[30]=0$, and $iparm[35]&gt;0$. To specify values that differ in $A$ for low rank update (see Low Rank Update), set $iparm[38]=1$. The size of the array must be at least $2*ndiff + 1$, where $ndiff$ is the number of values of $A$ that are different. The values of $perm$ should be: $perm = {ndiff, row_index1, column_index1, row_index2, column_index2, ..., row_index_ndiff, column_index_ndiff}$ where $row_index_m$ and $column_index_m$ are the row and column indices of the $m$-th differing non-zero value in matrix $A$. The row and column index pairs can be in any order, but must use zero-based indexing regardless of the value of $iparm[34]$.</td>
<td>in/out</td>
</tr>
<tr>
<td>Parameter</td>
<td>Type</td>
<td>Description</td>
<td>Values</td>
<td>Comments</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
</tbody>
</table>
| nrhs      | MKL_INT* | Number of right-hand sides that need to be solved for                       | >=0    | Generally used value is 1
To obtain better Intel® MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase. |
| iparm[64] | MKL_INT* | This array is used to pass various parameters to Intel® MKL PARDISO and to return some useful information after execution of the solver (see iparm Parameter for more details) | *      | If iparm[0]=0, Intel® MKL PARDISO fills iparm[1] through iparm[63] with default values and uses them.                                                                                                  |
| msglvl    | MKL_INT* | Message level information                                                   | 0      | Intel® MKL PARDISO generates no output                                                                                                       |
|           |          |                                                                             | 1      | Intel® MKL PARDISO prints statistical information                                                                                             |
| b[n*nrhs]| void*   | Right-hand side vectors                                                   | *      | On entry, contains the right-hand side vector/matrix B, which is placed contiguously in memory. The \( b[i+k*n] \) element must hold the \( i \)-th component of \( k \)-th right-hand side vector. Note that \( b \) is only accessed in the solution phase. On output, the array is replaced with the solution if iparm[5]=1. |
| x[n*nrhs]| void*   | Solution vectors                                                           | *      | On output, if iparm[5]=0, contains solution vector/matrix X which is placed contiguously in memory. The \( x[i+k*n] \) element                                                                      |

**NOTE**
Unless you have specified low rank update, iparm[34] indicates whether row/column indexing starts from 1 or 0.
### Parameter Types

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
<th>Values</th>
<th>Comments</th>
<th>In/Out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>must hold the ( i )-th component of ( k )-th solution vector. Note that ( x ) is only accessed in the solution phase.</td>
<td></td>
</tr>
</tbody>
</table>

**error** MKL_INT* Error indicator

- **0**: No error
- **-1**: Input inconsistent
- **-2**: Not enough memory
- **-3**: Reordering problem
- **-4**: Zero pivot, numerical factorization or iterative refinement problem
- **-5**: Unclassified (internal) error
- **-6**: Reordering failed (matrix types 11 and 13 only)
- **-7**: Diagonal matrix is singular
- **-8**: 32-bit integer overflow problem
- **-9**: Not enough memory for OOC
- **-10**: Problems with opening OOC temporary files
- **-11**: Read/write problems with the OOC data file

1) See description of PARDISO_DATA_TYPE in PARDISO_DATA_TYPE.

### pardiso iparm Parameter

This table describes all individual components of the Intel® MKL PARDISO iparm parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iparm[0]</td>
<td>Use default values.</td>
</tr>
<tr>
<td>input</td>
<td>0 iparm[1] - iparm[63] are filled with default values.</td>
</tr>
<tr>
<td></td>
<td>≠0 You must supply all values in components iparm[1] - iparm[63].</td>
</tr>
</tbody>
</table>

---

Sparse Solver Routines 5

1781
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Caution</strong></td>
<td>You can control the parallel execution of the solver by explicitly setting the <code>MKL_NUM_THREADS</code> environment variable. If fewer OpenMP threads are available than specified, the execution may slow down instead of speeding up. If <code>MKL_NUM_THREADS</code> is not defined, then the solver uses all available processors.</td>
</tr>
<tr>
<td>0</td>
<td>The minimum degree algorithm [Li99].</td>
</tr>
<tr>
<td>2*</td>
<td>The parallel (OpenMP) version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Intel® MKL PARDISO Phase 1 takes significant time.</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Setting <code>iparm[1] = 3</code> prevents the use of CNR mode (<code>iparm[33] &gt; 0</code>) because Intel® MKL PARDISO uses dynamic parallelism.</td>
</tr>
<tr>
<td><strong>K=0</strong></td>
<td>The factorization is always computed as required by <code>phase</code>.</td>
</tr>
<tr>
<td><strong>K=1</strong></td>
<td>CGS iteration replaces the computation of <code>LU</code>. The preconditioner is <code>LU</code> that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</td>
</tr>
<tr>
<td><strong>K=2</strong></td>
<td>CGS iteration for symmetric positive definite matrices replaces the computation of <code>LL^T</code>. The preconditioner is <code>LL^T</code> that was computed at a previous step (the first step or last step with a failure) in a sequence of solutions needed for identical sparsity patterns.</td>
</tr>
<tr>
<td>The value <code>L</code> controls the stopping criterion of the Krylov Subspace iteration: <code>\[ \varepsilon_{\text{CGS}} = 10^{-L} \]</code> is used in the stopping criterion `[</td>
<td></td>
</tr>
<tr>
<td>A maximum number of 150 iterations is fixed with the assumption that the iteration will converge before consuming half the factorization time. Intermediate convergence rates and residue excursions are checked and can terminate the iteration process. If <code>phase = 23</code>, then the factorization for a given <code>A</code> is automatically recomputed in cases where the Krylov Subspace iteration failed, and the corresponding direct solution is returned. Otherwise, the solution from the preconditioned Krylov Subspace iteration is returned. Using <code>phase = 33</code> results in an error message (<code>error = -4</code>) if the stopping criteria for the Krylov Subspace iteration can not be reached. More information on the failure can be obtained from <code>iparm[19]</code>.</td>
<td></td>
</tr>
<tr>
<td>The default is <code>iparm[3] = 0</code>, and other values are only recommended for an advanced user. <code>iparm[3]</code> must be greater than or equal to zero.</td>
<td></td>
</tr>
<tr>
<td>Examples:</td>
<td></td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>iparm[3]</td>
<td>Description</td>
</tr>
<tr>
<td>31</td>
<td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-3 for nonsymmetric matrices</td>
</tr>
<tr>
<td>61</td>
<td>LU-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for nonsymmetric matrices</td>
</tr>
<tr>
<td>62</td>
<td>LLT-preconditioned CGS iteration with a stopping criterion of 1.0E-6 for symmetric positive definite matrices</td>
</tr>
</tbody>
</table>

**iparm[4]** input

User permutation.

This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® MKL PARDISO.

This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^TAP$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the perm parameter.

**Caution**

You can only set one of iparm[4], iparm[30], and iparm[35], so be sure that the iparm[30] (partial solution) and the iparm[35] (Schur complement) parameters are 0 if you set iparm[4].

<table>
<thead>
<tr>
<th>0*</th>
<th>User permutation in the perm array is ignored.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intel® MKL PARDISO uses the user supplied fill-in reducing permutation from the perm array. iparm[1] is ignored.</td>
</tr>
</tbody>
</table>

**NOTE**


| 2         | Intel® MKL PARDISO returns the permutation vector computed at phase 1 in the perm array. |

**iparm[5]** input

Write solution on x.

**NOTE**

The array x is always used.

<table>
<thead>
<tr>
<th>0*</th>
<th>The array x contains the solution; right-hand side vector b is kept unchanged.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The solver stores the solution on the right-hand side b.</td>
</tr>
</tbody>
</table>

**iparm[6]** output

Number of iterative refinement steps performed.

Reports the number of iterative refinement steps that were actually performed during the solve step.

**iparm[7]** input

Iterative refinement step.

On entry to the solve and iterative refinement step, iparm[7] must be set to the maximum number of iterative refinement steps that the solver performs.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization.</td>
</tr>
</tbody>
</table>
| >0        | Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of $\text{iparm}[7]$ steps of iterative refinement. The solver might stop the process before the maximum number of steps if  
- a satisfactory level of accuracy of the solution in terms of backward error is achieved,  
- or if it determines that the required accuracy cannot be reached. In this case Intel® MKL PARDISO returns -4 in the error parameter.  
The number of executed iterations is reported in $\text{iparm}[6]$. |
| <0        | Same as above, but the accumulation of the residuum uses extended precision real and complex data types.  
Perturbed pivots result in iterative refinement (independent of $\text{iparm}[7]=0$) and the number of executed iterations is reported in $\text{iparm}[6]$. |
| $\text{iparm}[8]$ | Reserved. Set to zero. |
| $\text{iparm}[9]$ | Pivoting perturbation.  
This parameter instructs Intel® MKL PARDISO how to handle small pivots or zero pivots for nonsymmetric matrices ($\text{mtype}=11$ or $\text{mtype}=13$) and symmetric matrices ($\text{mtype}=-2$, $\text{mtype}=-4$, or $\text{mtype}=6$). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].  
Small pivots are perturbed with $\text{eps} = 10^{-\text{iparm}[9]}$.  
The magnitude of the potential pivot is tested against a constant threshold of  
$\text{alpha} = \text{eps} ||A2||_{\text{inf}}$,  
where $\text{eps} = 10^{-\text{iparm}[9]}$, $A2 = P^*F_{\text{vol}}^*D_c^*A^*D_c^*P$, and $||A2||_{\text{inf}}$ is the infinity norm of the scaled and permuted matrix $A$. Any tiny pivots encountered during elimination are set to the sign $||A1||\cdot\text{eps}||A2||_{\text{inf}}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $\text{eps} = 10^{-\text{iparm}[9]}$. |
| 13*       | The default value for nonsymmetric matrices ($\text{mtype}=11$, $\text{mtype}=13$), $\text{eps} = 10^{-13}$. |
| 8*        | The default value for symmetric indefinite matrices ($\text{mtype}=-2$, $\text{mtype}=-4$, $\text{mtype}=6$), $\text{eps} = 10^{-8}$. |
| $\text{iparm}[10]$ | Scaling vectors.  
Intel® MKL PARDISO uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less than or equal to 1. This scaling method is applied only to nonsymmetric matrices ($\text{mtype}=11$ or $\text{mtype}=13$). The scaling can also be used for symmetric indefinite matrices ($\text{mtype}=-2$, $\text{mtype}=-4$, or $\text{mtype}=6$) when the symmetric weighted matchings are applied ($\text{iparm}[12]=1$). |
## Sparse Solver Routines

<table>
<thead>
<tr>
<th><strong>Component</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Use $iparm[10] = 1$ (scaling) and $iparm[12] = 1$ (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase ($phase=11$) you must provide the numerical values of the matrix $A$ in array $a$ in case of scaling and symmetric weighted matching.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$iparm[11]$</th>
<th><strong>input</strong> Solve with transposed or conjugate transposed matrix $A$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE</strong> For real matrices, the terms <em>transposed</em> and <em>conjugate transposed</em> are equivalent.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$iparm[12]$</th>
<th><strong>input</strong> Improved accuracy using (non-) symmetric weighted matching.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® MKL PARDISO can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.</td>
<td></td>
</tr>
</tbody>
</table>

| $iparm[13]$ | **output** Number of perturbed pivots. After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6. |

<p>| $iparm[14]$ | <strong>output</strong> Peak memory on symbolic factorization. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iparm[15] output</td>
<td>The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase. This value is only computed in phase 1.</td>
</tr>
<tr>
<td>iparm[16] output</td>
<td>Permanent memory on symbolic factorization. Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases. This value is only computed in phase 1.</td>
</tr>
<tr>
<td>iparm[17] input/output</td>
<td>Report the number of non-zero elements in the factors.</td>
</tr>
<tr>
<td>iparm[18] input/output</td>
<td>Report number of floating point operations (in 10^6 floating point operations) that are necessary to factor the matrix A.</td>
</tr>
<tr>
<td>iparm[19] output</td>
<td>Report CG/CGS diagnostics.</td>
</tr>
</tbody>
</table>

**NOTE**

Use iparm[10] = 1 (scaling) and iparm[12] = 1 (matchings) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Apply 1x1 diagonal pivoting during the factorization process.</td>
</tr>
<tr>
<td>1*</td>
<td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of ( mtype=-2, mtype=-4 ), or ( mtype=6 ).</td>
</tr>
<tr>
<td>2</td>
<td>Apply 1x1 diagonal pivoting during the factorization process. Using this value is the same as using ( iparm[20] = 0 ) except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by ( iparm[7] ) (0 by default).</td>
</tr>
<tr>
<td>3</td>
<td>Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of ( mtype=-2, mtype=-4 ), or ( mtype=6 ). Using this value is the same as using ( iparm[20] = 1 ) except that the solve step does not automatically make iterative refinements when perturbed pivots are obtained during numerical factorization. The number of iterations is limited to the number of iterative refinements specified by ( iparm[7] ) (0 by default).</td>
</tr>
</tbody>
</table>

**iparm[21]**

| output | Inertia: number of positive eigenvalues. Intel® MKL PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices. |

**iparm[22]**

| output | Inertia: number of negative eigenvalues. Intel® MKL PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices. |

**iparm[23]**

<table>
<thead>
<tr>
<th>input</th>
<th>Parallel factorization control.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>Intel® MKL PARDISO uses the classic algorithm for factorization.</td>
</tr>
<tr>
<td>1</td>
<td>Intel® MKL PARDISO uses a two-level factorization algorithm. This algorithm generally improves scalability in case of parallel factorization on many OpenMP threads (more than eight).</td>
</tr>
</tbody>
</table>

**NOTE** Disable \( iparm[10] \) (scaling) and \( iparm[12] = 1 \) (matching) when using the two-level factorization algorithm. Otherwise Intel® MKL PARDISO uses the classic factorization algorithm.

| 10     | Intel® MKL PARDISO uses an improved two-level factorization algorithm for nonsymmetric matrices. |

**iparm[24]**

<table>
<thead>
<tr>
<th>input</th>
<th>Parallel forward/backward solve control.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>Intel® MKL PARDISO uses the following strategy for parallelizing the solving step: In the case of the one right-hand side, the parallelization will be performed by partitioning the matrix. Otherwise, the parallelization will be over the right-hand sides. This feature is available only for in-core Intel® MKL PARDISO (see ( iparm[59] )).</td>
</tr>
<tr>
<td>1</td>
<td>Intel® MKL PARDISO uses the sequential forward and backward solve.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>2</td>
<td>Independent from the number of the right-hand sides, Intel® MKL PARDISO uses the parallel algorithm based on the matrix partitioning. This feature is available only for in-core Intel® MKL PARDISO (see iparm[59]).</td>
</tr>
</tbody>
</table>

**iparm[25]**  
Reserved. Set to zero.

**iparm[26]**  
Matrix checker.  
- **0** Intel® MKL PARDISO does not check the sparse matrix representation for errors.  
- **1** Intel® MKL PARDISO checks integer arrays ia and ja. In particular, Intel® MKL PARDISO checks whether column indices are sorted in increasing order within each row.

**iparm[27]**  
Single or double precision Intel® MKL PARDISO.  
See iparm[7] for information on controlling the precision of the refinement steps.  
**Important** The iparm[27] value is stored in the Intel® MKL PARDISO handle between Intel® MKL PARDISO calls, so the precision mode can be changed only during phase 1.

- **0** Input arrays (a, x and b) and all internal arrays must be presented in double precision.  
- **1** Input arrays (a, x and b) must be presented in single precision. In this case all internal computations are performed in single precision.

**iparm[28]**  
Reserved. Set to zero.

**iparm[29]**  
Number of zero or negative pivots.  
If Intel® MKL PARDISO detects zero or negative pivot for mtype=2 or mtype=4 matrix types, the factorization is stopped. Intel® MKL PARDISO returns immediately with an error = -4, and iparm[29] reports the number of the equation where the zero or negative pivot is detected.  
**Note:** The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.

**iparm[30]**  
Partial solve and computing selected components of the solution vectors.  
This parameter controls the solve step of Intel® MKL PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define perm so that when perm(i) = 1 it means that either the i-th component in the right-hand sides is nonzero, or the i-th component in the solution vectors is computed, or both, depending on the value of iparm[30].  
The permutation vector perm must be present in all phases of Intel® MKL PARDISO software. At the reordering step, the software overwrites the input vector perm by a permutation vector used by the software at the factorization and solver step. If m is the number of components such that perm(i) = 1, then the last m components of the output vector perm are a set of the indices i satisfying the condition perm(i) = 1 on input.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOTE</strong></td>
<td>Turning on this option often increases the time used by Intel® MKL PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.</td>
</tr>
<tr>
<td><strong>Important</strong></td>
<td>You can use this feature for both in-core and out-of-core Intel® MKL PARDISO as long as iparm[23]=1. Otherwise, you cannot use partial solve for out-of-core mode and you will need to set iparm[59]=0 for in-core mode. Set the parameters iparm[7] (iterative refinement steps), iparm[3] (preconditioned CGS), iparm[4] (user permutation), and iparm[35] (Schur complement) to 0 as well.</td>
</tr>
</tbody>
</table>

| 0* | Disables this option. |
| 1 | It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector perm is defined so that perm(i) = 1 means that the (i)-th component in the right-hand sides is nonzero. In this case Intel® MKL PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the i-th component in the solution vectors is only computed if perm(i) = 1. |
| 2 | It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector perm is defined so that perm(i) = 1 means that the i-th component in the right-hand sides is nonzero. Unlike for iparm[30]=1, all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for iparm[30]=2 you must set the i-th component of the right-hand sides to zero explicitly if perm(i) is not equal to 1. |
| 3 | Selected components of the solution vectors are computed. The perm array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case perm(i) = 1 means that the i-th component in the solution vectors is computed. |

**iparm[31] - iparm[32]** | Reserved. Set to zero. |
| **iparm[33]** | Optimal number of OpenMP threads for conditional numerical reproducibility (CNR) mode. Intel® MKL PARDISO reads the value of iparm[33] during the analysis phase (phase 1), so you cannot change it later. |

Because Intel® MKL PARDISO uses C random number generator facilities during the analysis phase (phase 1) you must take these precautions to get numerically reproducible results:

- Do not alter the states of the random number generators.
- Do not run multiple instances of Intel® MKL PARDISO in parallel in the analysis phase (phase 1).
Component | Description
--- | ---

**NOTE**
CNR is only available for the in-core version of Intel® MKL PARDISO and the non-parallel version of the nested dissection algorithm. You must also:

- set `iparm[59]` to 0 in order to use the in-core version,
- not set `iparm[1]` to 3 in order to not use the parallel version of the nested dissection algorithm.

Otherwise Intel® MKL PARDISO does not produce numerically repeatable results even if CNR is enabled for Intel® MKL using the functionality described in Support Functions for CNR.

| 0* | CNR mode for Intel® MKL PARDISO is enabled only if it is enabled for Intel® MKL using the functionality described in Support Functions for CNR and the in-core version is used. Intel® MKL PARDISO determines the optimal number of OpenMP threads automatically, and produces numerically reproducible results regardless of the number of threads. |
| >0 | CNR mode is enabled for Intel® MKL PARDISO if in-core version is used and the optimal number of OpenMP threads for Intel® MKL PARDISO to rely on is defined by the value of `iparm[33]`. You can use `iparm[33]` to enable CNR mode independent from other Intel® MKL domains. To get the best performance, set `iparm[33]` to the actual number of hardware threads dedicated for Intel® MKL PARDISO. Setting `iparm[33]` to fewer OpenMP threads than the maximum number of them in use reduces the scalability of the problem being solved. Setting `iparm[33]` to more threads than are available can reduce the performance of Intel® MKL PARDISO. |

**iparm[34]**
input

One- or zero-based indexing of columns and rows.

**NOTE**
Schur complement may be inaccurate or incorrect if pivots are detected.
Please, check the output of `iparm[28]`.

| 0* | One-based indexing: columns and rows indexing in arrays `ia`, `ja`, and `perm` starts from 1 (Fortran-style indexing). |
| 1 | Zero-based indexing: columns and rows indexing in arrays `ia`, `ja`, and `perm` starts from 0 (C-style indexing). |

**iparm[35]**
input/output

Schur complement matrix computation control. To calculate this matrix, you must set the input permutation vector `perm` to a set of indexes such that when `perm(i) = 1`, the `i`-th element of the initial matrix is an element of the Schur matrix.

**Caution**

<p>| 0* | Do not compute Schur complement. |
| 1 | Compute Schur complement matrix as part of Intel® MKL PARDISO factorization step and return it in the solution vector. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Compute Schur complement matrix as part of Intel® MKL PARDISO factorization step and return it in the solution vector. Since this option calculates factorization arrays you can use it to launch partial or full solution of the entire problem after the factorization step.</td>
</tr>
<tr>
<td>-1</td>
<td>Same as iparm[35] equals 1, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with pardiso_export. After reordering stage of MKL PARDISO, iparm[35] contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase.</td>
</tr>
<tr>
<td>-2</td>
<td>Same as iparm[35] equals 2, but the Schur complement matrix is provided in 3-array CSR sparse format. Use in combination with pardiso_export. After reordering stage of MKL PARDISO, iparm[35] contains number of nonzero elements for Schur complement matrix. Set it once again before calling the factorization phase.</td>
</tr>
</tbody>
</table>

**iparm[36]**

Format for matrix storage.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>Use CSR format (see Three Array Variation of CSR Format) for matrix storage.</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>Use BSR format (see Three Array Variation of BSR Format) for matrix storage with blocks of size iparm[36].</td>
</tr>
</tbody>
</table>

**NOTE**

Intel® MKL does not support BSR format in these cases:

- iparm[10] > 0: Scaling vectors
- iparm[12] > 0: Weighted matching
- iparm[30] > 0: Partial solution
- iparm[35] > 0: Schur complement
- iparm[55] > 0: Pivoting control
- iparm[59] > 0: OOC Intel® MKL PARDISO

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>Convert supplied matrix to variable BSR (VBSR) format (see Sparse Data Storage) for matrix storage. Intel® MKL PARDISO analyzes the matrix provided in CSR3 format and converts it to an internal VBSR format. Set iparm[36] = -t, 0 &lt; t ≤ 100.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Intel® MKL supports only the VBSR format for real and symmetric positive definite or indefinite matrices ((mtype = 2) or (mtype = -2)). Intel® MKL does not support VBSR format in these cases:</td>
</tr>
<tr>
<td></td>
<td>- (iparm[10] &gt; 0): Scaling vectors</td>
</tr>
<tr>
<td></td>
<td>- (iparm[12] &gt; 0): Weighted matching</td>
</tr>
<tr>
<td></td>
<td>- (iparm[55] &gt; 0): Pivoting control</td>
</tr>
<tr>
<td><strong>NOTE</strong></td>
<td>Intel® MKL supports these features for all matrix types as long as (iparm[23] = 1):</td>
</tr>
<tr>
<td></td>
<td>- (iparm[30] &gt; 0): Partial solution</td>
</tr>
<tr>
<td></td>
<td>- (iparm[35] &gt; 0): Schur complement</td>
</tr>
<tr>
<td></td>
<td>- (iparm[59] &gt; 0): OOC Intel® MKL PARDISO</td>
</tr>
</tbody>
</table>

| \(iparm[37]\) | Reserved. Set to zero. |
| \(iparm[38]\) | Enable low rank update (see Low Rank Update) to accelerate factorization for multiple matrices with identical structure and similar values. |
| 0* | Do not use low rank update functionality. |
| 1 | Use low rank update functionality. You must also set \(iparm[23] = 10\) and provide a list of changed values in the \(perm\) array. |
| | This option requires the default settings of \(iparm[3], iparm[4], iparm[5], iparm[27], iparm[30], iparm[35], iparm[36], iparm[55],\) and \(iparm[59]\) as well. |

| \(iparm[39] - iparm[41]\) | Reserved. Set to zero. |
| \(iparm[42]\) | Control parameter for the computation of the diagonal of inverse matrix. |
| 0* | Do not compute the diagonal of inverse matrix. |
| 1 | Intel® MKL PARDISO computes the diagonal of the inverse matrix during the factorization phase. This feature is only available with two-level factorization algorithm (\(iparm[23] = 1\)) and real symmetric matrices (\(mtype = 2\) or \(mtype = -2\)). The diagonal is returned in the solution vector. |

| \(iparm[43] - iparm[54]\) | Reserved. Set to zero. |
| \(iparm[55]\) | Diagonal and pivoting control. |
| 0* | Internal function used to work with pivot and calculation of diagonal arrays turned off. |
| 1 | You can use the \texttt{mkl\_pardiso\_pivot} callback routine to control pivot elements which appear during numerical factorization. Additionally, you can obtain the elements of initial matrix and factorized matrices after the \texttt{pardiso} factorization step diagonal using the \texttt{pardiso\_getdiag} routine. This parameter can be turned on only in the in-core version of Intel® MKL PARDISO. |

<p>| (iparm[56] - iparm[58]) | Reserved. Set to zero. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iparm[59]</td>
<td>Intel® MKL PARDISO mode. iparm[59] switches between in-core (IC) and out-of-core (OOC) Intel® MKL PARDISO. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC. Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better Intel® MKL PARDISO performance by setting iparm[59] before the reordering phase. The amount of memory used in OOC mode depends on the number of OpenMP threads.</td>
</tr>
<tr>
<td>NOTE</td>
<td>When iparm[59] &gt; 0, use the MKL_PARDISO_OOC_FILE_NAME environment variable to store factors.</td>
</tr>
<tr>
<td>Warning</td>
<td>Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.</td>
</tr>
<tr>
<td>0*</td>
<td>IC mode.</td>
</tr>
<tr>
<td>1</td>
<td>IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables: MKL_PARDISO_OOC_MAX_CORE_SIZE (default value 2000 MB) and MKL_PARDISO_OOC_MAX_SWAP_SIZE (default value 0 MB); otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of MKL_PARDISO_OOC_MAX_CORE_SIZE. If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.</td>
</tr>
<tr>
<td>NOTE</td>
<td>Conditional numerical reproducibility (CNR) is not supported for this mode.</td>
</tr>
<tr>
<td>2</td>
<td>OOC mode. The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode. If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible. To obtain better Intel® MKL PARDISO performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase. Refer to How to use Intel® MKL OOC PARDISO and Storage of Matrices for more details about OOC.</td>
</tr>
<tr>
<td>iparm[60] - iparm[61]</td>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>Component</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>iparm[62]</td>
<td>Size of the minimum OOC memory for numerical factorization and solution. This parameter provides the size in kilobytes of the minimum memory required by OOC Intel® MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel® MKL PARDISO can be estimated as (\max(iparm[14], iparm[15] + iparm[62])).</td>
</tr>
<tr>
<td>iparm[63]</td>
<td>Reserved. Set to zero.</td>
</tr>
</tbody>
</table>

**NOTE**
Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See Sparse Matrix Storage Formats for an example.

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**PARDISO_DATA_TYPE**

The following table lists the values of `PARDISO_DATA_TYPE` depending on the matrix types and values of the parameter `iparm[27].`

<table>
<thead>
<tr>
<th>Data type value</th>
<th>Matrix type mtype</th>
<th>iparm[27]</th>
<th>comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>1, 2, -2, 11</td>
<td>0</td>
<td>Real matrices, double precision</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>1</td>
<td>Real matrices, single precision</td>
</tr>
<tr>
<td>MKL_Complex16</td>
<td>3, 6, 13, 4, -4</td>
<td>0</td>
<td>Complex matrices, double precision</td>
</tr>
<tr>
<td>MKL_Complex8</td>
<td></td>
<td>1</td>
<td>Complex matrices, single precision</td>
</tr>
</tbody>
</table>

**Parallel Direct Sparse Solver for Clusters Interface**

The Parallel Direct Sparse Solver for Clusters Interface solves large linear systems of equations with sparse matrices on clusters. It is

- high performing
robust  
memory efficient  
easy to use

A hybrid implementation combines Message Passing Interface (MPI) technology for data exchange between parallel tasks (processes) running on different nodes, and OpenMP* technology for parallelism inside each node of the cluster. This approach effectively uses modern hardware resources such as clusters consisting of nodes with multi-core processors. The solver code is optimized for the latest Intel processors, but also performs well on clusters consisting of non-Intel processors.

Code examples are available in the Intel® MKL installation examples directory.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

**Parallel Direct Sparse Solver for Clusters Interface Algorithm**

Parallel Direct Sparse Solver for Clusters Interface solves a set of sparse linear equations

\[ A^*X = B \]

with multiple right-hand sides using a distributed \( LU \), \( LL^T \), \( LDL^T \) or \( LDL^* \) factorization, where \( A \) is an \( n \times n \) matrix, and \( X \) and \( B \) are \( n \times nrhs \) matrices.

The solution comprises four tasks:

- analysis and symbolic factorization;
- numerical factorization;
- forward and backward substitution including iterative refinement;
- termination to release all internal solver memory.

The solver first computes a symmetric fill-in reducing permutation \( P \) based on the nested dissection algorithm from the METIS package [Karypis98](included with Intel® MKL), followed by the Cholesky or other type of factorization (depending on matrix type) [Schenk00-2] of \( PAP^T \). The solver uses either diagonal pivoting, or 1x1 and 2x2 Bunch and Kaufman pivoting for symmetric indefinite or Hermitian matrices before finding an approximation of \( X \) by forward and backward substitution and iterative refinement.

The initial matrix \( A \) is perturbed whenever numerically acceptable 1x1 and 2x2 pivots cannot be found within the diagonal blocks. One or two passes of iterative refinement may be required to correct the effect of the perturbations. This restricted notion of pivoting with iterative refinement is effective for highly indefinite symmetric systems. For a large set of matrices from different application areas, the accuracy of this method is comparable to a direct factorization method that uses complete sparse pivoting techniques [Schenk04].

Parallel Direct Sparse Solver for Clusters additionally improves the pivoting accuracy by applying symmetric weighted matching algorithms. These methods identify large entries in the coefficient matrix \( A \) that, if permuted close to the diagonal, enable the factorization process to identify more acceptable pivots and proceed with fewer pivot perturbations. The methods are based on maximum weighted matching and improve the quality of the factor in a complementary way to the alternative idea of using more complete pivoting techniques.
Parallel Direct Sparse Solver for Clusters Interface Matrix Storage

The sparse data storage in the Parallel Direct Sparse Solver for Clusters Interface follows the scheme described in the Sparse Matrix Storage Formats section using the variable $ja$ for columns, $ia$ for rowIndex, and $a$ for values. Column indices $ja$ must be in increasing order per row.

When an input data structure is not accessed in a call, a NULL pointer or any valid address can be passed as a placeholder for that argument.

Algorithm Parallelization and Data Distribution

Intel® MKL Parallel Direct Sparse Solver for Clusters enables parallel execution of the solution algorithm with efficient data distribution.

The master MPI process performs the symbolic factorization phase to represent matrix $A$ as computational tree. Then matrix $A$ is divided among all MPI processes in a one-dimensional manner. The same distribution is used for $L$-factor (the lower triangular matrix in Cholesky decomposition). Matrix $A$ and all required internal data are broadcast to slave MPI processes. Each MPI process fills in its own parts of $L$-factor with initial values of the matrix $A$.

Parallel Direct Sparse Solver for Clusters Interface computes all independent parts of $L$-factor completely in parallel. When a block of the factor must be updated by other blocks, these updates are independently passed to a temporary array on each updating MPI process. It further gathers the result into an updated block using the MPI_Reduce() routine. The computations within an MPI process are dynamically divided among OpenMP threads using pipelining parallelism with a combination of left- and right-looking techniques similar to those of the PARDISO* software. Level 3 BLAS operations from Intel® MKL ensure highly efficient performance of block-to-block update operations.

During forward/backward substitutions, respective Right Hand Side (RHS) parts are distributed among all MPI processes. All these processes participate in the computation of the solution. Finally, the solution is gathered on the master MPI process.

This approach demonstrates good scalability on clusters with Infiniband* technology. Another advantage of the approach is the effective distribution of $L$-factor among cluster nodes. This enables the solution of tasks with a much higher number of non-zero elements than it is possible with any Symmetric Multiprocessing (SMP) in-core direct solver.

The algorithm ensures that the memory required to keep internal data on each MPI process is decreased when the number of MPI processes in a run increases. However, the solver requires that matrix $A$ and some other internal arrays completely fit into the memory of each MPI process.

To get the best performance, run one MPI process per physical node and set the number of OpenMP* threads per node equal to the number of physical cores on the node.

**NOTE**

Instead of calling MPI_Init(), initialize MPI with MPI_Init_thread() and set the MPI threading level to MPI_THREAD_FUNNELED or higher. For details, see the code examples in <install_dir>/examples.

cluster_sparse_solver

*Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides.*
Syntax

```c
void cluster_sparse_solver (_MKL_DSS_HANDLE_t pt, const MKL_INT *maxfct, const MKL_INT *mnum, const MKL_INT *mtype, const MKL_INT *phase, const MKL_INT *n, const void *a, const MKL_INT *ia, const MKL_INT *ja, MKL_INT *perm, const MKL_INT *nrhs, MKL_INT *iparm, const MKL_INT *msglvl, void *b, void *x, const int *comm, MKL_INT *error);
```

Include Files

- mkl_cluster_sparse_solver.h

Description

The routine `cluster_sparse_solver` calculates the solution of a set of sparse linear equations

\[ A \times X = B \]

with single or multiple right-hand sides, using a parallel \( LU \), \( LDL \), or \( LL^T \) factorization, where \( A \) is an \( n \)-by-\( n \) matrix, and \( X \) and \( B \) are \( n \)-by-\( nrhs \) vectors or matrices.

**NOTE**

This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

**NOTE**

Most of the input parameters (except for the `pt`, `phase`, and `comm` parameters and, for the distributed format, the `a`, `ia`, and `ja` arrays) must be set on the master MPI process only, and ignored on other processes. Other MPI processes get all required data from the master MPI process using the MPI communicator, `comm`.

- **pt**
  - Array of size 64.
  - Handle to internal data structure. The entries must be set to zero before the first call to `cluster_sparse_solver`.
  - **Caution**
    After the first call to `cluster_sparse_solver` do not modify `pt`, as that could cause a serious memory leak.

- **maxfct**
  - Ignored; assumed equal to 1.

- **mnum**
  - Ignored; assumed equal to 1.

- **mtype**
  - Defines the matrix type, which influences the pivoting method. The Parallel Direct Sparse Solver for Clusters solver supports the following matrices:

```plaintext
1    real and structurally symmetric
2    real and symmetric positive definite
-2   real and symmetric indefinite
3    complex and structurally symmetric
```
<table>
<thead>
<tr>
<th>phase</th>
<th>Solver Execution Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Analysis</td>
</tr>
<tr>
<td>12</td>
<td>Analysis, numerical factorization</td>
</tr>
<tr>
<td>13</td>
<td>Analysis, numerical factorization, solve, iterative refinement</td>
</tr>
<tr>
<td>22</td>
<td>Numerical factorization</td>
</tr>
<tr>
<td>23</td>
<td>Numerical factorization, solve, iterative refinement</td>
</tr>
<tr>
<td>33</td>
<td>Solve, iterative refinement</td>
</tr>
<tr>
<td>-1</td>
<td>Release all internal memory for all matrices</td>
</tr>
</tbody>
</table>

Number of equations in the sparse linear systems of equations $A^*X = B$. Constraint: $n > 0$.

Array. Contains the non-zero elements of the coefficient matrix $A$ corresponding to the indices in $ja$. The coefficient matrix can be either real or complex. The matrix must be stored in the three-array variant of the compressed sparse row (CSR3) or in the three-array variant of the block compressed sparse row (BSR3) format, and the matrix must be stored with increasing values of $ja$ for each row.

For CSR3 format, the size of $a$ is the same as that of $ja$. Refer to the values array description in Three Array Variation of CSR Format for more details.

For BSR3 format the size of $a$ is the size of $ja$ multiplied by the square of the block size. Refer to the values array description in Three Array Variation of BSR Format for more details.
NOTE
For centralized input (iparm[39]=0), provide the $a$ array for the master MPI process only. For distributed assembled input (iparm[39]=1 or iparm[39]=2), provide it for all MPI processes.

Important
The column indices of non-zero elements of each row of the matrix $A$ must be stored in increasing order.

For CSR3 format, $ia[i]$ ($i<n$) points to the first column index of row $i$ in the array $ja$. That is, $ia[i]$ gives the index of the element in array $a$ that contains the first non-zero element from row $i$ of $A$. The last element $ia[n]$ is taken to be equal to the number of non-zero elements in $A$, plus one. Refer to $rowIndex$ array description in Three Array Variation of CSR Format for more details.

For BSR3 format, $ia[i]$ ($i<n$) points to the first column index of row $i$ in the array $ja$. That is, $ia[i]$ gives the index of the element in array $a$ that contains the first non-zero block from row $i$ of $A$. The last element $ia[n]$ is taken to be equal to the number of non-zero blocks in $A$, plus one. Refer to $rowIndex$ array description in Three Array Variation of BSR Format for more details.

The array $ia$ is accessed in all phases of the solution process.

Indexing of $ia$ is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter iparm[34]. For zero-based indexing, the last element $ia[n]$ is assumed to be equal to the number of non-zero elements in matrix $A$.

NOTE
For centralized input (iparm[39]=0), provide the $ia$ array at the master MPI process only. For distributed assembled input (iparm[39]=1 or iparm[39]=2), provide it at all MPI processes.

For CSR3 format, array $ja$ contains column indices of the sparse matrix $A$. It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for $columns$ array in Three Array Variation of CSR Format.

For BSR3 format, array $ja$ contains column indices of the sparse matrix $A$. It is important that the indices are in increasing order per row. For symmetric matrices, the solver needs only the upper triangular part of the system as is shown for $columns$ array in Three Array Variation of BSR Format.

The array $ja$ is accessed in all phases of the solution process.

Indexing of $ja$ is one-based by default, but it can be changed to zero-based by setting the appropriate value to the parameter iparm[35].
**NOTE**

For centralized input \((\text{iparm}(40)=0)\), provide the \(ja\) array at the master MPI process only. For distributed assembled input \((\text{iparm}(40)=1\) or \(\text{iparm}(40)=2)\), provide it at all MPI processes.

*perm*

Ignored.

*nrhs*

Number of right-hand sides that need to be solved for.

*iparm*

Array, size 64. This array is used to pass various parameters to Parallel Direct Sparse Solver for Clusters Interface and to return some useful information after execution of the solver.

See [cluster_sparse_solver iparm Parameter](#) for more details about the *iparm* parameters.

*msglvl*

Message level information. If \(\text{msglvl} = 0\) then `cluster_sparse_solver` generates no output, if \(\text{msglvl} = 1\) the solver prints statistical information to the screen.

Statistics include information such as the number of non-zero elements in \(L\)-factor and the timing for each phase.

Set \(\text{msglvl} = 1\) if you report a problem with the solver, since the additional information provided can facilitate a solution.

*b*

Array, size \(n*nrhs\). On entry, contains the right-hand side vector/matrix \(B\), which is placed in memory contiguously. The \(b[i+k*n]\) must hold the \(i\)-th component of \(k\)-th right-hand side vector. Note that \(b\) is only accessed in the solution phase.

*comm*

MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the `MPI_Comm_c2f()` function. See the examples in the `<install_dir>/examples` directory.

**Output Parameters**

*pt*

Handle to internal data structure.

Ignored.

*perm*

On output, some *iparm* values report information such as the numbers of non-zero elements in the factors.

See [cluster_sparse_solver iparm Parameter](#) for more details about the *iparm* parameters.

*b*

On output, the array is replaced with the solution if \(\text{iparm}[5] = 1\).

*x*

Array, size \((n*nrhs)\). If \(\text{iparm}[5]=0\) it contains solution vector/matrix \(X\), which is placed contiguously in memory. The \(x[i+k*n]\) element must hold the \(i\)-th component of the \(k\)-th solution vector. Note that \(x\) is only accessed in the solution phase.

*error*

The error indicator according to the below table:
### Error Information

<table>
<thead>
<tr>
<th>Error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>input inconsistent</td>
</tr>
<tr>
<td>-2</td>
<td>not enough memory</td>
</tr>
<tr>
<td>-3</td>
<td>reordering problem</td>
</tr>
<tr>
<td>-4</td>
<td>Zero pivot, numerical factorization or iterative refinement problem. If the error appears during the solution phase, try to change the pivoting perturbation (iparm[9]) and also increase the number of iterative refinement steps. If it does not help, consider changing the scaling, matching and pivoting options (iparm[10], iparm[12], iparm[20])</td>
</tr>
<tr>
<td>-5</td>
<td>unclassified (internal) error</td>
</tr>
<tr>
<td>-6</td>
<td>reordering failed (matrix types 11 and 13 only)</td>
</tr>
<tr>
<td>-7</td>
<td>diagonal matrix is singular</td>
</tr>
<tr>
<td>-8</td>
<td>32-bit integer overflow problem</td>
</tr>
<tr>
<td>-9</td>
<td>not enough memory for OOC</td>
</tr>
<tr>
<td>-10</td>
<td>error opening OOC files</td>
</tr>
<tr>
<td>-11</td>
<td>read/write error with OOC files</td>
</tr>
</tbody>
</table>

### cluster_sparse_solver_64

** Calculates the solution of a set of sparse linear equations with single or multiple right-hand sides. **

### Syntax

```c
void cluster_sparse_solver_64 ( _MKL_DSS_HANDLE_t pt, const long long int *maxfct, const long long int *mnum, const long long int *mtype, const long long int *phase, const long long int *n, const void *a, const long long int *ia, const long long int *ja, long long int *perm, const long long int *nrhs, long long int *iparm, const long long int *msglvl, void *b, void *x, const int *comm, long long int *error);
```

### Include Files

- mkl_cluster_sparse_solver.h

### Description

The routine `cluster_sparse_solver_64` is an alternative ILP64 (64-bit integer) version of the `cluster_sparse_solver` routine (see the Description section for more details). The interface of `cluster_sparse_solver_64` is the same as the interface of `cluster_sparse_solver`, but it accepts and returns all integer data as long long int.

Use `cluster_sparse_solver_64` when `cluster_sparse_solver` for solving large matrices (with the number of non-zero elements on the order of 500 million or more). You can use it together with the usual LP64 interfaces for the rest of Intel® MKL functionality. In other words, if you use 64-bit integer version...
(cluster_sparse_solver_64), you do not need to re-link your applications with ILP64 libraries. Take into account that cluster_sparse_solver_64 may perform slower than regular cluster_sparse_solver on the reordering and symbolic factorization phase.

**NOTE**
cluster_sparse_solver_64 is supported only in the 64-bit libraries. If cluster_sparse_solver_64 is called from the 32-bit libraries, it returns error =-12.

**NOTE**
This routine supports the Progress Routine feature. See Progress Function section for details.

**Input Parameters**
The input parameters of cluster_sparse_solver_64 are the same as the input parameters of cluster_sparse_solver, but cluster_sparse_solver_64 accepts all integer data as long long int.

**Output Parameters**
The output parameters of cluster_sparse_solver_64 are the same as the output parameters of cluster_sparse_solver, but cluster_sparse_solver_64 returns all integer data as long long int.

**cluster_sparse_solver_get_csr_size**
Computes the (local) number of rows and (local) number of nonzero entries for (distributed) CSR data corresponding to the provided name.

**Syntax**
void cluster_sparse_solver_get_csr_size (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_DATA name, MKL_INT *local_nrows, MKL_INT *local_nnz, const int*comm, MKL_INT *error);

**Include Files**
- mkl_cluster_sparse_solver.h

**Description**
This routine uses the internal data created during the factorization phase of cluster_sparse_solver for matrix A. The routine then:
- Computes the local number of rows and the local number of nonzeros for CSR data that correspond to the provided name.
- Returns the computed values in local_nrows and local_nnz.

It is assumed that the CSR data defined by the name will be distributed in the same way as the matrix A (as defined by iparm[39]) used in cluster_sparse_solver.

The returned values can be used for allocating CSR arrays for factors L and U, and also for allocating arrays for permutations P and Q, or scaling matrix D which can then be used with cluster_sparse_solver_set_csr_ptrs or cluster_sparse_solver_set_ptr for exporting corresponding data via cluster_sparse_solver_export.
NOTE
Only call this routine after the factorization phase (\texttt{phase}=22) of the \texttt{cluster_sparse_solver} has been called. Neither \texttt{pt}, nor \texttt{iparm} should be changed after the preceding call to \texttt{cluster_sparse_solver}.

Input Parameters

\texttt{pt}  
Array with size of 64.  
Handle to internal data structure used in the prior calls to \texttt{cluster_sparse_solver}.

\texttt{Caution}
Do not modify \texttt{pt} after the calls to \texttt{cluster_sparse_solver}.

\texttt{name}  
Specifies CSR data for which the output values are computed.

\begin{itemize}
  \item \texttt{SPARSE\_PTLUQT\_L}: Factor \texttt{L} from \texttt{P\*A\*Q=LU}.  
  \item \texttt{SPARSE\_PTLUQT\_U}: Factor \texttt{U} from \texttt{P\*A\*Q=LU}.  
  \item \texttt{SPARSE\_DPTLUQT\_L}: Factor \texttt{L} from \texttt{P\*(D^{-1}\*A)*Q=LU}.  
  \item \texttt{SPARSE\_DPTLUQT\_U}: Factor \texttt{U} from \texttt{P\*(D^{-1}\*A)*Q=LU}.  
\end{itemize}

\texttt{local\_nrows}  
On entry, an array of size 1.

\texttt{local\_nnz}  
On entry, an array of size 1.

\texttt{comm}  
MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the \texttt{MPI\_Comm\_c2f()} function. See the examples in the <install\_dir>/examples directory.

Output Parameters

\texttt{local\_nrows}  
On output, the local number of rows for the CSR data which correspond to the \texttt{name}.

\texttt{local\_nnz}  
On output, the local number of nonzero entries for the CSR data which correspond to the \texttt{name}.

\texttt{error}  
The error indicator:

\begin{itemize}
  \item \texttt{error}: Information
    \begin{itemize}
      \item 0: no error
      \item -1: \texttt{pt} is a null pointer
      \item -2: invalid \texttt{pt}
      \item -3: invalid \texttt{name}
      \item -4: unsupported \texttt{name}
    \end{itemize}
\end{itemize}
**error** | **Information** 
---|---
-9 | unsupported internal code path, consider switching off non-default iparm parameters for cluster_sparse_solver
-10 | unsupported case when the matrix A is distributed among processes with overlap in the preceding calls to cluster_sparse_solver
-12 | internal memory error

**NOTE** Refer to cl_solver_export_c.c for an example using this functionality.

**cluster_sparse_solver_set_csr_ptrs**

* Saves internally-provided pointers to the 3-array CSR data corresponding to the specified name. 

**Syntax**

```c
void cluster_sparse_solver_set_csr_ptrs (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_DATA name, MKL_INT *rowptr, MKL_INT *colindx, void *vals, const int *comm, MKL_INT *error);
```

**Include Files**

- mkl_cluster_sparse_solver.h

**Description**

This routine internally saves the input pointers, rowptr, colindx, and vals, of the 3-array CSR data, which correspond to the provided name. It is assumed that the exported data will be distributed in the same way as the matrix A (as defined by iparm[39]) used in cluster_sparse_solver.

The saved pointers can then be used for exporting corresponding data by means of cluster_sparse_solver_export.

**NOTE**

Only call this routine after the factorization phase (phase=22) of the cluster_sparse_solver has been called. Neither pt, nor iparm should be changed after the preceding call to cluster_sparse_solver.

**Input Parameters**

**pt**

Array with size of 64.

Handle to internal data structure used in the prior calls to cluster_sparse_solver.

**Caution**

Do not modify pt after the calls to cluster_sparse_solver.

**name**

Specifies for which CSR data the pointers are provided.
Sparse Solver Routines

SPARSE_PTLUQT_L  Factor $L$ from $PAQ=L^*U$.
SPARSE_PTLUQT_U  Factor $U$ from $PAQ=L^*U$.
SPARSE_DPTLUQT_L  Factor $L$ from $(D^{-1}A)Q=L^*U$.
SPARSE_DPTLUQT_U  Factor $U$ from $(D^{-1}A)Q=L^*U$.

rowptr

Array of length at least $(\text{local\_nrows}+1)$ where $\text{local\_nrows}$ is the local number of rows, which can be obtained by calling \text{cluster\_sparse\_solver\_get\_csr\_size}. This array contains row indices, such that $\text{rowptr}[i] \cdot \text{indexing}$ is the first index of row $i$ in the array’s $\text{vals}$ and $\text{colindx}$. Here, the value of $\text{indexing}$ is 0 for zero-based indexing and 1 for one-based indexing, and must be the same as it was for the matrix $A$ used in the preceding calls to \text{cluster\_sparse\_solver} (also stored in $\text{iparm}[34]$).

Refer to $\text{pointerB}$ array description in \text{CSR Format} for more details.

colindx

Array of length at least $\text{rowptr}[\text{local\_nrows}]-\text{rowptr}[0]$. Indexing (zero- or one-based) must be the same as for $\text{rowptr}$. For one-based indexing, the array contains the column indices plus one for each non-zero element of the matrix which corresponds to the $\text{name}$. For zero-based indexing, the array contains the column indices for each non-zero element of the matrix.

vals

Array containing non-zero elements of the matrix which corresponds to the $\text{name}$. Its length is equal to length of the $\text{colindx}$ array. Refer to values array description in \text{CSR Format} for more details.

It will be interpreted internally as float*/double*/MKL\_Complex8*/MKL\_Complex16* depending on $\text{mtype}$ (type of the matrix $A$) and $\text{iparm}[27]$ (precision) specified in the preceding call to \text{cluster\_sparse\_solver}.

comm

MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the \text{MPI\_Comm\_c2f()} function. See the examples in the $\langle\text{install\_dir}\rangle$/examples directory.

Output Parameters

error

The error indicator:

<table>
<thead>
<tr>
<th>error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>$pt$ is a null pointer</td>
</tr>
<tr>
<td>-2</td>
<td>invalid $pt$</td>
</tr>
<tr>
<td>-3</td>
<td>invalid $\text{name}$</td>
</tr>
<tr>
<td>-4</td>
<td>unsupported $\text{name}$</td>
</tr>
</tbody>
</table>
**error**

-9  unsupported internal code path, consider switching off non-default *iparm* parameters for *cluster_sparse_solver*

-12  internal memory error

**NOTE** Refer to *cl_solver_export_c.c* for an example using this functionality.

---

**cluster_sparse_solver_set_ptr**

*Internally saves a provided pointer to the data corresponding to the specified name.*

**Syntax**

```c
void cluster_sparse_solver_set_ptr (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_DATA name, void *ptr, const int *comm, MKL_INT *error);
```

**Include Files**

- `mkl_cluster_sparse_solver.h`

**Description**

This routine internally saves the input pointer, *ptr*, of the data which correspond to the provided *name*. The saved pointer can then be used for exporting corresponding data by means of *cluster_sparse_solver_export*.

**NOTE**

Only call this routine after the factorization phase (*phase=22*) of the *cluster_sparse_solver* has been called. Neither *pt*, nor *iparm* should be changed after the preceding call to *cluster_sparse_solver*.

**Input Parameters**

**pt**

Array with size of 64.

Handle to internal data structure used in the prior calls to *cluster_sparse_solver*.

**Caution**

Do not modify *pt* after the calls to *cluster_sparse_solver*.

**name**

Specifies the data for which the pointer is provided.

- `SPARSE_PTLUQT_P`  Permutation *P* from *P*A*Q* = *L*U.
- `SPARSE_PTLUQT_Q`  Permutation *Q* from *P*A*Q* = *L*U.
- `SPARSE_DPTLUQT_P`  Permutation *P* from *P* (D⁻¹*A) *Q* = *L*U.
Permutation $Q$ from $P^*(D^{-1}A)*Q=L*U$.

Scaling (diagonal) $D$ from $P^*(D^{-1}A)*Q=L*U$.

vals

Array containing elements of the vector representation for the data which corresponds to the name. Its length should be at least $\text{local}_nrows$, where $\text{local}_nrows$ is the local number of rows in a corresponding matrix (obtained from $\text{cluster\_sparse\_solver\_get\_csr\_size}$, for example).

For permutations $P$ and $Q$, $\text{vals}$ is interpreted as MKL_INT*, while for the scaling it is interpreted as float*/double*(MKL_Complex8*/MKL_Complex16*) depending on $\text{mtype}$ (type of the matrix $A$) and $\text{iparm}[27]$ (precision) specified in the preceding call to $\text{cluster\_sparse\_solver}$.

comm

MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the $\text{MPI\_Comm\_c2f()}$ function. See the examples in the $<$install_dir$/>$examples directory.

Output Parameters

error

The error indicator:

<table>
<thead>
<tr>
<th>error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>$pt$ is a null pointer</td>
</tr>
<tr>
<td>-2</td>
<td>invalid $pt$</td>
</tr>
<tr>
<td>-3</td>
<td>invalid name</td>
</tr>
<tr>
<td>-4</td>
<td>unsupported name</td>
</tr>
<tr>
<td>-9</td>
<td>unsupported internal code path, consider switching off non-default $iparm$ parameters for $\text{cluster_sparse_solver}$</td>
</tr>
<tr>
<td>-12</td>
<td>internal memory error</td>
</tr>
</tbody>
</table>

**NOTE** Refer to $\text{cl\_solver\_export\_c\_c}$ for an example using this functionality.

---

$\text{cluster\_sparse\_solver\_export}$

Computes data corresponding to the specified decomposition (defined by export operation) and fills the pointers provided by calls to $\text{cluster\_sparse\_solver\_set\_ptr}$ and/or $\text{cluster\_sparse\_solver\_set\_csr\_ptrs}$.

**Syntax**

```c
void cluster_sparse_solver_export (_MKL_DSS_HANDLE_t pt, _MKL_DSS_EXPORT_OPERATION operation, const int *comm, MKL_INT *error);
```
Include Files

- mkl_cluster_sparse_solver.h

Description

This routine computes the data for the pointers of the (distributed) data to be exported (as defined the specified operation). It is assumed that the exported data will be distributed in the same way as the matrix \( A \) (as defined by \( \text{iparm[39]} \)) used in \( \text{cluster_sparse_solver} \).

**NOTE**
Only call this routine after the factorization phase (\( \text{phase}=22 \)) of the \( \text{cluster_sparse_solver} \) has been called. Neither \( pt \), nor \( \text{iparm} \) should be changed after the preceding call to \( \text{cluster_sparse_solver} \).

**NOTE**
Only call this routine after all pointers to the data required for the specified operation have been provided by means of calling \( \text{cluster_sparse_solver_set_ptr} \) and/or \( \text{cluster_sparse_solver_set_csr_ptrs} \).

Input Parameters

\( pt \)

Array with size of 64.

Handle to internal data structure used in the prior calls to \( \text{cluster_sparse_solver} \).

**Caution**
Do not modify \( pt \) after the calls to \( \text{cluster_sparse_solver} \).

\( operation \)

Specifies a particular operation which defines what data are exported

- \( \text{SPARSE_PTLUQT} \)
  
  Exporting data from decomposition \( P*A*Q=L*U \).

- \( \text{SPARSE_DPTLUQT} \)
  
  Exporting data from decomposition from \( P*(D^{-1}A)*Q=L*U \).

**NOTE**
Currently, for \( operation=\text{SPARSE_DPTLUQT} \) a real (complex) unit vector is provided for the scaling matrix \( D \). Do not turn on scaling(\( \text{iparm[10]} > 0 \)) or matching(\( \text{iparm[12]} > 0 \)) in the \( \text{iparm} \) during the call to \( \text{cluster_sparse_solver} \) for this value of \( operation \).

\( comm \)

MPI communicator. The solver uses the Fortran MPI communicator internally. Convert the MPI communicator to Fortran using the \( \text{MPI_Comm_c2f()} \) function. See the examples in the \(<\text{install_dir}/\text{examples}\) directory.
Output Parameters

The error indicator:

<table>
<thead>
<tr>
<th>Error</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no error</td>
</tr>
<tr>
<td>-1</td>
<td>( pt ) is a null pointer</td>
</tr>
<tr>
<td>-2</td>
<td>invalid ( pt )</td>
</tr>
<tr>
<td>-5</td>
<td>invalid operation</td>
</tr>
<tr>
<td>-6</td>
<td>pointers to some of the data required for the specified operation were not provided prior to calling ( \text{cluster_sparse_solver_export} )</td>
</tr>
<tr>
<td>-9</td>
<td>unsupported internal code path, consider switching off non-default ( iparm ) parameters for ( \text{cluster_sparse_solver} )</td>
</tr>
<tr>
<td>-10</td>
<td>unsupported case when the matrix ( A ) is distributed among processes with overlap in the preceding calls to ( \text{cluster_sparse_solver} )</td>
</tr>
<tr>
<td>-12</td>
<td>internal memory error</td>
</tr>
</tbody>
</table>

NOTE Refer to \( \text{cl\_solver\_export\_c\_c} \) for an example using this functionality.

\textbf{cluster\_sparse\_solver} \textit{iparm} Parameter

The following table describes all individual components of the Parallel Direct Sparse Solver for Clusters Interface \textit{iparm} parameter. Components which are not used must be initialized with 0. Default values are denoted with an asterisk (*).

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( iparm[0] ) input</td>
<td>Use default values.</td>
</tr>
<tr>
<td>0</td>
<td>( iparm[1] - iparm(64) ) are filled with default values.</td>
</tr>
<tr>
<td>!=0</td>
<td>You must supply all values in components ( iparm[1] - iparm(64) ).</td>
</tr>
<tr>
<td>( iparm[1] ) input</td>
<td>Fill-in reducing ordering for the input matrix.</td>
</tr>
<tr>
<td>2*</td>
<td>The nested dissection algorithm from the METIS package [Karypis98].</td>
</tr>
<tr>
<td>3</td>
<td>The parallel version of the nested dissection algorithm. It can decrease the time of computations on multi-core computers, especially when Phase 1 takes significant time.</td>
</tr>
<tr>
<td>10</td>
<td>The MPI version of the nested dissection and symbolic factorization algorithms. The input matrix for the reordering must be distributed among different MPI processes without any intersection. Use ( iparm[40] ) and ( iparm[41] ) to set the bounds of the domain. During all of Phase 1, the entire matrix is not gathered on any one process, which can decrease computation time (especially when Phase 1 takes significant time) and decrease memory usage for each MPI process on the cluster.</td>
</tr>
</tbody>
</table>
### Component Description

**NOTE**
If you set `iparm[1] = 10`, `comm = -1` (MPI communicator), and if there is one MPI process, optimization and full parallelization with the OpenMP version of the nested dissection and symbolic factorization algorithms proceeds. This can decrease computation time on multi-core computers. In this case, set `iparm[40] = 1` and `iparm[41] = n` for one-based indexing, or to 0 and `n - 1`, respectively, for zero-based indexing.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>iparm[2]</strong></td>
<td>Reserved. Set to zero.</td>
</tr>
</tbody>
</table>
| **iparm[4]** | User permutation.  
This parameter controls whether user supplied fill-in reducing permutation is used instead of the integrated multiple-minimum degree or nested dissection algorithms. Another use of this parameter is to control obtaining the fill-in reducing permutation vector calculated during the reordering stage of Intel® MKL PARDISO.  
This option is useful for testing reordering algorithms, adapting the code to special applications problems (for instance, to move zero diagonal elements to the end of $P^*A^*P^T$), or for using the permutation vector more than once for matrices with identical sparsity structures. For definition of the permutation, see the description of the `perm` parameter.  
**Caution**  
| 0 | User permutation in the `perm` array is ignored.  
1 | Intel® MKL PARDISO uses the user supplied fill-in reducing permutation from the `perm` array. `iparm[1]` is ignored.  
2 | Intel® MKL PARDISO returns the permutation vector computed at phase 1 in the `perm` array. |
| **iparm[5]** | Write solution on $x$.  
**NOTE**  
The array $x$ is always used.  
| 0* | The array $x$ contains the solution; right-hand side vector $b$ is kept unchanged.  
1 | The solver stores the solution on the right-hand side $b$. |
| **iparm[6]** | Number of iterative refinement steps performed.  
Reports the number of iterative refinement steps that were actually performed during the solve step. |
| **iparm[7]** | Iterative refinement step.  
On entry to the solve and iterative refinement step, `iparm[7]` must be set to the maximum number of iterative refinement steps that the solver performs.  
<p>| 0* | The solver automatically performs two steps of iterative refinement when perturbed pivots are obtained during the numerical factorization. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
</table>
| $>0$      | Maximum number of iterative refinement steps that the solver performs. The solver performs not more than the absolute value of $iparm[7]$ steps of iterative refinement. The solver might stop the process before the maximum number of steps if  
  - a satisfactory level of accuracy of the solution in terms of backward error is achieved,  
  - or if it determines that the required accuracy cannot be reached. In this case Parallel Direct Sparse Solver for Clusters Interface returns $-4$ in the $error$ parameter.  
  
The number of executed iterations is reported in $iparm[6]$. |
| $<0$      | Same as above, but the accumulation of the residuum uses extended precision real and complex data types.  
  
Perturbed pivots result in iterative refinement (independent of $iparm[7]=0$) and the number of executed iterations is reported in $iparm[6]$. |
| $iparm[8]$| Reserved. Set to zero. |
  This parameter instructs Parallel Direct Sparse Solver for Clusters Interface how to handle small pivots or zero pivots for nonsymmetric matrices ($mtype = 11$ or $mtype = 13$) and symmetric matrices ($mtype = -2, mtype = -4,$ or $mtype = 6$). For these matrices the solver uses a complete supernode pivoting approach. When the factorization algorithm reaches a point where it cannot factor the supernodes with this pivoting strategy, it uses a pivoting perturbation strategy similar to [Li99], [Schenk04].  
  
Small pivots are perturbed with $eps = 10^{-iparm[9]}$.  
  
The magnitude of the potential pivot is tested against a constant threshold of $alpha = eps \cdot ||A2||_{\infty}$,  
  
where $eps = 10^{-iparm[9]}$, $A2 = P \cdot P_{WS} \cdot D_{r} \cdot A \cdot D_{c} \cdot P$, and $||A2||_{\infty}$ is the infinity norm of the scaled and permuted matrix $A$. Any tiny pivots encountered during elimination are set to the sign $(\frac{l}{1}) \cdot eps \cdot ||A2||_{\infty}$, which trades off some numerical stability for the ability to keep pivots from getting too small. Small pivots are therefore perturbed with $eps = 10^{-iparm[9]}$. |
| 13*      | The default value for nonsymmetric matrices ($mtype = 11, mtype=13$), $eps = 10^{-13}$. |
| 8*       | The default value for symmetric indefinite matrices ($mtype = -2, mtype=-4,$ $mtype=6$), $eps = 10^{-8}$. |
  Parallel Direct Sparse Solver for Clusters Interface uses a maximum weight matching algorithm to permute large elements on the diagonal and to scale.  
  
Use $iparm[10] = 1$ (scaling) and $iparm[12] = 1$ (matching) for highly indefinite symmetric matrices, for example, from interior point optimizations or saddle point problems. Note that in the analysis phase ($phase=11$) you must provide the numerical values of the matrix $A$ in array $a$ in case of scaling and symmetric weighted matching. |
<p>| 0*       | Disable scaling. Default for symmetric indefinite matrices. |
| 1*       | Enable scaling. Default for nonsymmetric matrices. |</p>
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale the matrix so that the diagonal elements are equal to 1 and the absolute values of the off-diagonal entries are less or equal to 1. This scaling method is applied to nonsymmetric matrices ($mtype = 11$, $mtype = 13$). The scaling can also be used for symmetric indefinite matrices ($mtype = -2$, $mtype = -4$, $mtype = 6$) when the symmetric weighted matchings are applied ($iparm[12] = 1$). Note that in the analysis phase (phase=11) you must provide the numerical values of the matrix $A$ in case of scaling.</td>
<td></td>
</tr>
</tbody>
</table>

**iparm[11]** Solve with transposed or conjugate transposed matrix $A$.

**NOTE**
For real matrices, the terms *transposed* and *conjugate transposed* are equivalent.

0* Solve a linear system $AX = B$.
1 Solve a conjugate transposed system $A^H X = B$ based on the factorization of the matrix $A$.
2 Solve a transposed system $A^T X = B$ based on the factorization of the matrix $A$.

**iparm[12]** Improved accuracy using (non-) symmetric weighted matching.
Parallel Direct Sparse Solver for Clusters Interface can use a maximum weighted matching algorithm to permute large elements close the diagonal. This strategy adds an additional level of reliability to the factorization methods and complements the alternative of using more complete pivoting techniques during the numerical factorization.

0* Disable matching. Default for symmetric indefinite matrices.
1* Enable matching. Default for nonsymmetric matrices.

Maximum weighted matching algorithm to permute large elements close to the diagonal.

It is recommended to use $iparm[10] = 1$ (scaling) and $iparm[12] = 1$ (matching) for highly indefinite symmetric matrices, for example from interior point optimizations or saddle point problems.

Note that in the analysis phase (phase=11) you must provide the numerical values of the matrix $A$ in case of symmetric weighted matching.

**iparm[13]** Number of perturbed pivots.
After factorization, contains the number of perturbed pivots for the matrix types: 1, 3, 11, 13, -2, -4 and 6.

**iparm[14]** Peak memory on symbolic factorization.
The total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.
This value is only computed in phase 1.

**iparm[15]** Permanent memory on symbolic factorization.
Permanent memory from the analysis and symbolic factorization phase in kilobytes that the solver needs in the factorization and solve phases.
This value is only computed in phase 1.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iparm[16]</td>
<td>Size of factors/Peak memory on numerical factorization and solution. This parameter provides the size in kilobytes of the total memory consumed by in-core Intel® MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. See iparm[62] for the OOC mode. The total peak memory consumed by Intel® MKL PARDISO is max(iparm[14], iparm[15]+iparm[16])</td>
</tr>
<tr>
<td>iparm[17]</td>
<td>Report the number of non-zero elements in the factors. &lt;0 Enable reporting if iparm[17] &lt; 0 on entry. The default value is -1. &gt;=0 Disable reporting.</td>
</tr>
<tr>
<td>iparm[18]</td>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>iparm[19]</td>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>iparm[20]</td>
<td>Pivoting for symmetric indefinite matrices. 0 Apply 1x1 diagonal pivoting during the factorization process. 1* Apply 1x1 and 2x2 Bunch-Kaufman pivoting during the factorization process. Bunch-Kaufman pivoting is available for matrices of mtype=-2, mtype=-4, or mtype=6.</td>
</tr>
<tr>
<td>iparm[21]</td>
<td>Inertia: number of positive eigenvalues. Intel® MKL PARDISO reports the number of positive eigenvalues for symmetric indefinite matrices.</td>
</tr>
<tr>
<td>iparm[22]</td>
<td>Inertia: number of negative eigenvalues. Intel® MKL PARDISO reports the number of negative eigenvalues for symmetric indefinite matrices.</td>
</tr>
<tr>
<td>iparm[23]</td>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>iparm[26]</td>
<td>Matrix checker. 0* Do not check the sparse matrix representation for errors. 1 Check integer arrays ia and ja. In particular, check whether the column indices are sorted in increasing order within each row.</td>
</tr>
<tr>
<td>iparm[27]</td>
<td>Single or double precision Parallel Direct Sparse Solver for Clusters Interface. See iparm[7] for information on controlling the precision of the refinement steps. 0* Input arrays (a, x and b) and all internal arrays must be presented in double precision. 1 Input arrays (a, x and b) must be presented in single precision. In this case all internal computations are performed in single precision.</td>
</tr>
<tr>
<td>iparm[28]</td>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>iparm[29]</td>
<td>Number of zero or negative pivots.</td>
</tr>
</tbody>
</table>
If Intel® MKL PARDISO detects zero or negative pivot for \( mtype = 2 \) or \( mtype = 4 \) matrix types, the factorization is stopped. Intel® MKL PARDISO returns immediately with an error \(-4\), and \( iparm[29] \) reports the number of the equation where the zero or negative pivot is detected.

Note: The returned value can be different for the parallel and sequential version in case of several zero/negative pivots.

### \( iparm[30] \)
- **Input**
- Partial solve and computing selected components of the solution vectors.

This parameter controls the solve step of Intel® MKL PARDISO. It can be used if only a few components of the solution vectors are needed or if you want to reduce the computation cost at the solve step by utilizing the sparsity of the right-hand sides. To use this option the input permutation vector define \( perm \) so that when \( perm(i) = 1 \) it means that either the \( i \)-th component in the right-hand sides is nonzero, or the \( i \)-th component in the solution vectors is computed, or both, depending on the value of \( iparm[30] \).

The permutation vector \( perm \) must be present in all phases of Intel® MKL PARDISO software. At the reordering step, the software overwrites the input vector \( perm \) by a permutation vector used by the software at the factorization and solver step. If \( m \) is the number of components such that \( perm(i) = 1 \), then the last \( m \) components of the output vector \( perm \) are a set of the indices \( i \) satisfying the condition \( perm(i) = 1 \) on input.

**NOTE**
- Turning on this option often increases the time used by Intel® MKL PARDISO for factorization and reordering steps, but it can reduce the time required for the solver step.

**Important**
- Set the parameters \( iparm[7] \) (iterative refinement steps), \( iparm[3] \) (preconditioned CGS), \( iparm[4] \) (user permutation), and \( iparm[35] \) (Schur complement) to 0 as well.

<table>
<thead>
<tr>
<th>( iparm[30] )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>0</strong> * Disables this option.</td>
</tr>
<tr>
<td><strong>1</strong></td>
</tr>
<tr>
<td>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector ( perm ) is defined so that ( perm(i) = 1 ) means that the ( (i) )-th component in the right-hand sides is nonzero. In this case Intel® MKL PARDISO only uses the non-zero components of the right-hand side vectors and computes only corresponding components in the solution vectors. That means the ( i )-th component in the solution vectors is only computed if ( perm(i) = 1 ).</td>
</tr>
<tr>
<td><strong>2</strong></td>
</tr>
<tr>
<td>It is assumed that the right-hand sides have only a few non-zero components* and the input permutation vector ( perm ) is defined so that ( perm(i) = 1 ) means that the ( i )-th component in the right-hand sides is nonzero. Unlike for ( iparm[30]=1 ), all components of the solution vector are computed for this setting and all components of the right-hand sides are used. Because all components are used, for ( iparm[30]=2 ) you must set the ( i )-th component of the right-hand sides to zero explicitly if ( perm(i) ) is not equal to 1.</td>
</tr>
<tr>
<td><strong>3</strong></td>
</tr>
<tr>
<td>Selected components of the solution vectors are computed. The ( perm ) array is not related to the right-hand sides and it only indicates which components of the solution vectors should be computed. In this case ( perm(i) = 1 ) means that the ( i )-th component in the solution vectors is computed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( iparm[30] ), ( iparm[32] )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved. Set to zero.</td>
</tr>
<tr>
<td>Component</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>iparm[34]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0*</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>iparm[35]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0*</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>iparm[36]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0*</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>&lt; 0</td>
</tr>
<tr>
<td>iparm[37]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>iparm[38]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>iparm[39]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Component</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>0*</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

\( iparm[40] \) input
Beginning of input domain.
The number of the matrix \( A \) row, RHS element, and, for \( iparm[39]=2 \), solution vector that begins the input domain belonging to this MPI process.
Only applicable to the distributed assembled matrix input format (\( iparm[39] > 0 \)).
See Sparse Matrix Storage Formats for more details.

\( iparm[41] \) input
End of input domain.
The number of the matrix \( A \) row, RHS element, and, for \( iparm[39]=2 \), solution vector that ends the input domain belonging to this MPI process.
Only applicable to the distributed assembled matrix input format (\( iparm[39] > 0 \)).
See Sparse Matrix Storage Formats for more details.

\( iparm[42] \), \( iparm[58] \) input
Reserved. Set to zero.

\( iparm[59] \) input
cluster_sparse_solver mode.
\( iparm[59] \) switches between in-core (IC) and out-of-core (OOC) of cluster_sparse Solver. OOC can solve very large problems by holding the matrix factors in files on the disk, which requires a reduced amount of main memory compared to IC.
Unless you are operating in sequential mode, you can switch between IC and OOC modes after the reordering phase. However, you can get better cluster_sparse_solver performance by setting \( iparm[59] \) before the reordering phase.
The amount of memory used in OOC mode depends on the number of OpenMP threads.

**NOTE**
When \( iparm[59] > 0 \), use the MKL_PARDISO_OOC_FILE_NAME environment variable to store factors.
<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Warning</strong></td>
<td>Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.</td>
</tr>
</tbody>
</table>

0* IC mode.

1 IC mode is used if the total amount of RAM (in megabytes) needed for storing the matrix factors is less than sum of two values of the environment variables:

- MKL_PARDISO_OOC_MAX_CORE_SIZE (default value 2000 MB)
- MKL_PARDISO_OOC_MAX_SWAP_SIZE (default value 0 MB)

otherwise OOC mode is used. In this case amount of RAM used by OOC mode cannot exceed the value of MKL_PARDISO_OOC_MAX_CORE_SIZE.

If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.

**NOTE**
Conditional numerical reproducibility (CNR) is not supported for this mode.

2 OOC mode.

The OOC mode can solve very large problems by holding the matrix factors in files on the disk. Hence the amount of RAM required by OOC mode is significantly reduced compared to IC mode.

If the total peak memory needed for storing the local arrays is more than MKL_PARDISO_OOC_MAX_CORE_SIZE, increase MKL_PARDISO_OOC_MAX_CORE_SIZE if possible.

To obtain better cluster_sparse_solver performance, during the numerical factorization phase you can provide the maximum number of right-hand sides, which can be used further during the solving phase.

**NOTE** To use OOC mode, you must disable iparm[10] (scaling) and iparm[12] = 1 (matching).

<table>
<thead>
<tr>
<th>iparm[60]</th>
<th>Reserved. Set to zero.</th>
</tr>
</thead>
<tbody>
<tr>
<td>iparm[61]</td>
<td>input</td>
</tr>
<tr>
<td>iparm[62]</td>
<td>output Size of the minimum OOC memory for numerical factorization and solution. This parameter provides the size in kilobytes of the minimum memory required by OOC Intel® MKL PARDISO for internal floating point arrays. This parameter is computed in phase 1. Total peak memory consumption of OOC Intel® MKL PARDISO can be estimated asmax(iparm[14], iparm[15] + iparm[62]).</td>
</tr>
<tr>
<td>iparm[63]</td>
<td>input Reserved. Set to zero.</td>
</tr>
</tbody>
</table>
NOTE
Generally in sparse matrices, components which are equal to zero can be considered non-zero if necessary. For example, in order to make a matrix structurally symmetric, elements which are zero can be considered non-zero. See Sparse Matrix Storage Formats for an example.

Optimization Notice
Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Direct Sparse Solver (DSS) Interface Routines
Intel® MKL supports the DSS interface, an alternative to the Intel® MKL PARDISO interface for the direct sparse solver. The DSS interface implements a group of user-callable routines that are used in the step-by-step solving process and utilizes the general scheme described in Appendix A Linear Solvers Basics for solving sparse systems of linear equations. This interface also includes one routine for gathering statistics related to the solving process.

The DSS interface also supports the out-of-core (OOC) mode.

Table "DSS Interface Routines" lists the names of the routines and describes their general use.

### DSS Interface Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dss_create</td>
<td>Initializes the solver and creates the basic data structures necessary for the solver. This routine must be called before any other DSS routine.</td>
</tr>
<tr>
<td>dss_define_structure</td>
<td>Informs the solver of the locations of the non-zero elements of the matrix.</td>
</tr>
<tr>
<td>dss_reorder</td>
<td>Based on the non-zero structure of the matrix, computes a permutation vector to reduce fill-in during the factoring process.</td>
</tr>
<tr>
<td>dss_factor_real, dss_factor_complex</td>
<td>Computes the $LU$, $LDL^T$ or $LL^T$ factorization of a real or complex matrix.</td>
</tr>
<tr>
<td>dss_solve_real, dss_solve_complex</td>
<td>Computes the solution vector for a system of equations based on the factorization computed in the previous phase.</td>
</tr>
<tr>
<td>dss_delete</td>
<td>Deletes all data structures created during the solving process.</td>
</tr>
<tr>
<td>dss_statistics</td>
<td>Returns statistics about various phases of the solving process.</td>
</tr>
</tbody>
</table>

To find a single solution vector for a single system of equations with a single right-hand side, invoke the Intel® MKL DSS interface routines in this order:

1. dss_create
2. dss_define_structure
3. dss_reorder
4. dss_factor_real, dss_factor_complex
5. dss_solve_real, dss_solve_complex
6. dss_delete

However, in certain applications it is necessary to produce solution vectors for multiple right-hand sides for a given factorization and/or factor several matrices with the same non-zero structure. Consequently, it is sometimes necessary to invoke the Intel® MKL sparse routines in an order other than that listed, which is possible using the DSS interface. The solving process is conceptually divided into six phases. Figure "Typical order for invoking DSS interface routines" indicates the typical order in which the DSS interface routines can be invoked.

Typical order for invoking DSS interface routines

Create
↓
Define array structure
↓
Reorder
↓
Factor
↓
Solve
↓
Delete

See the code examples that use the DSS interface routines to solve systems of linear equations in the Intel® MKL installation directory (dss_*.c).

• examples/solverc/source

DSS Interface Description

Each DSS routine reads from or writes to a data object called a handle. Refer to Memory Allocation and Handles to determine the correct method for declaring a handle argument for each language. For simplicity, the descriptions in DSS routines refer to the data type as MKL_DSS_HANDLE.

Routine Options

The DSS routines have an integer argument (referred below to as opt) for passing various options to the routines. The permissible values for opt should be specified using only the symbol constants defined in the language-specific header files (see Implementation Details). The routines accept options for setting the
message and termination levels as described in Table "Symbolic Names for the Message and Termination Levels Options". Additionally, each routine accepts the option MKL_DSS_DEFAULTS that sets the default values (as documented) for opt to the routine.

### Symbolic Names for the Message and Termination Levels Options

<table>
<thead>
<tr>
<th>Message Level</th>
<th>Termination Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKL_DSS_MSG_LVL_SUCCESS</td>
<td>MKL_DSS_TERM_LVL_SUCCESS</td>
</tr>
<tr>
<td>MKL_DSS_MSG_LVL_INFO</td>
<td>MKL_DSS_TERM_LVL_INFO</td>
</tr>
<tr>
<td>MKL_DSS_MSG_LVL_WARNING</td>
<td>MKL_DSS_TERM_LVL_WARNING</td>
</tr>
<tr>
<td>MKL_DSS_MSG_LVL_ERROR</td>
<td>MKL_DSS_TERM_LVL_ERROR</td>
</tr>
<tr>
<td>MKL_DSS_MSG_LVL_FATAL</td>
<td>MKL_DSS_TERM_LVL_FATAL</td>
</tr>
</tbody>
</table>

The settings for message and termination levels can be set on any call to a DSS routine. However, once set to a particular level, they remain at that level until they are changed in another call to a DSS routine.

You can specify both message and termination level for a DSS routine by adding the options together. For example, to set the message level to debug and the termination level to error for all the DSS routines, use the following call:

```c
  dss_create( handle, MKL_DSS_MSG_LVL_INFO + MKL_DSS_TERM_LVL_ERROR)
```

### User Data Arrays

Many of the DSS routines take arrays of user data as input. For example, pointers to user arrays are passed to the routine `dss_define_structure` to describe the location of the non-zero entries in the matrix.

**Caution**

Do not modify the contents of these arrays after they are passed to one of the solver routines.

### DSS Implementation Details

To promote portability across platforms and ease of use across different languages, use the `mkl_dss.h` header file.

The header file defines symbolic constants for returned error values, function options, certain defined data types, and function prototypes.

**NOTE**

Constants for options, returned error values, and message severities must be referred only by the symbolic names that are defined in these header files. Use of the Intel® MKL DSS software without including one of the above header files is not supported.

### Memory Allocation and Handles

You do not need to allocate any temporary working storage in order to use the Intel® MKL DSS routines, because the solver itself allocates any required storage. To enable multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using `ahandle` data object.
Each of the Intel® MKL DSS routines creates, uses, or deletes a handle. Consequently, any program calling an Intel® MKL DSS routine must be able to allocate storage for a handle. The exact syntax for allocating storage for a handle varies from language to language. To standardize the handle declarations, the language-specific header files declare constants and defined data types that must be used when declaring a handle object in your code.

```c
#include "mkl_dss.h"
_MKL_DSS_HANDLE_t handle;
```

In addition to the definition for the correct declaration of a handle, the include file also defines the following:

- function prototypes for languages that support prototypes
- symbolic constants that are used for the returned error values
- user options for the solver routines
- constants indicating message severity.

**DSS Routines**

**dss_create**

*Initializes the solver.*

**Syntax**

```c
MKL_INT dss_create(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt)
```

**Include Files**

- mkl.h

**Description**

The `dss_create` routine initializes the solver. After the call to `dss_create`, all subsequent invocations of the Intel® MKL DSS routines must use the value of the handle returned by `dss_create`.

**WARNING**

Do not write the value of handle directly.

The default value of the parameter `opt` is

`MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

By default, the DSS routines use double precision for solving systems of linear equations. The precision used by the DSS routines can be set to single mode by adding the following value to the `opt` parameter:

`MKL_DSS_SINGLE_PRECISION`.

Input data and internal arrays are required to have single precision.

By default, the DSS routines use Fortran style (one-based) indexing for input arrays of integer types (the first value is referenced as array element 1). To set indexing to C style (the first value is referenced as array element 0), add the following value to the `opt` parameter:

`MKL_DSS_ZERO_BASED_INDEXING`.

The `opt` parameter can also control number of refinement steps used on the solution stage by specifying the two following values:

- `MKL_DSS_REFINEMENT_OFF` - maximum number of refinement steps is set to zero;
- `MKL_DSS_REFINEMENT_ON` (default value) - maximum number of refinement steps is set to 2.
By default, DSS uses in-core computations. To launch the out-of-core version of DSS (OOC DSS) you can add to this parameter one of two possible values: MKL_DSS_OOC_STRONG and MKL_DSS_OOC_VARIABLE.

MKL_DSS_OOC_STRONG - OOC DSS is used.

MKL_DSS_OOC_VARIABLE - if the memory needed for the matrix factors is less than the value of the environment variable MKL_PARDISO_OOC_MAX_CORE_SIZE, then the OOC DSS uses the in-core kernels of Intel® MKL PARDISO, otherwise it uses the OOC computations.

The variable MKL_PARDISO_OOC_MAX_CORE_SIZE defines the maximum size of RAM allowed for storing work arrays associated with the matrix factors. It is ignored if MKL_DSS_OOC_STRONG is set. The default value of MKL_PARDISO_OOC_MAX_CORE_SIZE is 2000 MB. This value and default path and file name for storing temporary data can be changed using the configuration file pardiso_ooc.cfg or command line (See more details in the description of the pardiso routine).

**WARNING**
Other than message and termination level options, do not change the OOC DSS settings after they are specified in the routine dss_create.

### Input Parameters

**opt**
Parameter to pass the DSS options. The default value is MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR.

### Output Parameters

**handle**
Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

### Return Values

MKL_DSS_SUCCESS
MKL_DSS_INVALID_OPTION
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

**dss_define_structure**
*Communicates locations of non-zero elements in the matrix to the solver.*

### Syntax

```c
MKL_INT dss_define_structure( _MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, MKL_INT const *rowIndex, MKL_INT const *nRows, MKL_INT const *nCols, MKL_INT const *columns, MKL_INT const *nNonZeros);
```

### Include Files

- mkl.h
**Description**

The routine `dss_define_structure` communicates the locations of the `nNonZeros` number of non-zero elements in a matrix of `nRows * nCols` size to the solver.

---

**NOTE**

The Intel® MKL DSS software operates only on square matrices, so `nRows` must be equal to `nCols`.

To communicate the locations of non-zero elements in the matrix, do the following:

1. Define the general non-zero structure of the matrix by specifying the value for the options argument `opt`. You can set the following values for real matrices:
   - MKL_DSS_SYMMETRIC_STRUCTURE
   - MKL_DSS_SYMMETRIC
   - MKL_DSS_NON_SYMMETRIC
   and for complex matrices:
   - MKL_DSS_SYMMETRIC_STRUCTURE_COMPLEX
   - MKL_DSS_SYMMETRIC_COMPLEX
   - MKL_DSS_NON_SYMMETRIC_COMPLEX
   The information about the matrix type must be defined in `dss_define_structure`.
2. Provide the actual locations of the non-zeros by means of the arrays `rowIndex` and `columns` (see Sparse Matrix Storage Format).

---

**NOTE** No diagonal element can be omitted from the values array. If there is a zero value on the diagonal, for example, that element nonetheless must be explicitly represented.

---

**Input Parameters**

- `opt`  
  Parameter to pass the DSS options. The default value for the matrix structure is MKL_DSS_SYMMETRIC.

- `rowIndex`  
  Array of size `nRows+1`. Defines the location of non-zero entries in the matrix.

- `nRows`  
  Number of rows in the matrix.

- `nCols`  
  Number of columns in the matrix; must be equal to `nRows`.

- `columns`  
  Array of size `nNonZeros`. Defines the column location of non-zero entries in the matrix.

- `nNonZeros`  
  Number of non-zero elements in the matrix.

**Output Parameters**

- `handle`  
  Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

**Return Values**

- MKL_DSS_SUCCESS
MKL_DSS_STATE_ERR
MKL_DSS_INVALID_OPTION
MKL_DSS_STRUCTURE_ERR
MKL_DSS_ROW_ERR
MKL_DSS_COL_ERR
MKL_DSS_NOT_SQUARE
MKL_DSS_TOO_FEW_VALUES
MKL_DSS_TOO_MANY_VALUES
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

**dss_reorder**

*Computes or sets a permutation vector that minimizes the fill-in during the factorization phase.*

**Syntax**

```c
MKL_INT dss_reorder(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, MKL_INT const *perm)
```

**Include Files**

- mkl.h

**Description**

If `opt` contains the option `MKL_DSS_AUTO_ORDER`, then the routine `dss_reorder` computes a permutation vector that minimizes the fill-in during the factorization phase. For this option, the routine ignores the contents of the `perm` array.

If `opt` contains the option `MKL_DSS_METIS_OpenMP_ORDER`, then the routine `dss_reorder` computes permutation vector using the parallel nested dissections algorithm to minimize the fill-in during the factorization phase. This option can be used to decrease the time of `dss_reorder` call on multi-core computers. For this option, the routine ignores the contents of the `perm` array.

If `opt` contains the option `MKL_DSS_MY_ORDER`, then you must supply a permutation vector in the array `perm`. In this case, the array `perm` is of length `nRows`, where `nRows` is the number of rows in the matrix as defined by the previous call to `dss_define_structure`.

If `opt` contains the option `MKL_DSS_GET_ORDER`, then the permutation vector computed during the `dss_reorder` call is copied to the array `perm`. In this case you must allocate the array `perm` beforehand. The permutation vector is computed in the same way as if the option `MKL_DSS_AUTO_ORDER` is set.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
Input Parameters

opt

Parameter to pass the DSS options. The default value for the permutation type is MKL_DSS_AUTO_ORDER.

perm

Array of length nRows. Contains a user-defined permutation vector (accessed only if opt contains MKL_DSS_MY_ORDER or MKL_DSS_GET_ORDER).

Output Parameters

handle

Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).

Return Values

MKL_DSS_SUCCESS
MKL_DSS_STATE_ERR
MKL_DSS_INVALID_OPTION
MKL_DSS_REORDER_ERR
MKL_DSS_REORDER1_ERR
MKL_DSS_I32BIT_ERR
MKL_DSS_FAILURE
MKL_DSS_OUT_OF_MEMORY
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR

dss_factor_real, dss_factor_complex

Compute factorization of the matrix with previously specified location of non-zero elements.

Syntax

MKL_INT dss_factor_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *rValues)

MKL_INT dss_factor_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *cValues)

Include Files

• mkl.h
Description
These routines compute factorization of the matrix whose non-zero locations were previously specified by a call to `dss_define_structure` and whose non-zero values are given in the array `rValues`, `cValues` or `Values`. Data type These arrays must be of length `nNonZeros` as defined in a previous call to `dss_define_structure`.

**NOTE**
The data type (single or double precision) of `rValues`, `cValues`, `Values` must be in correspondence with precision specified by the parameter `opt` in the routine `dss_create`.

The `opt` argument can contain one of the following options:

- `MKL_DSS_POSITIVE_DEFINITE`
- `MKL_DSS_INDEFINITE`
- `MKL_DSS_HERMITIAN_POSITIVE_DEFINITE`
- `MKL_DSS_HERMITIAN_INDEFINITE`

depending on your matrix's type.

**NOTE**
This routine supports the Progress Routine feature. See Progress Function section for details.

Input Parameters

- `handle` pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
- `opt` parameter to pass the DSS options. The default value is MKL_DSS_POSITIVE_DEFINITE.
- `rValues` array of elements of the matrix A. Real data, single or double precision as it is specified by the parameter `opt` in the routine `dss_create`.
- `cValues` array of elements of the matrix A. Complex data, single or double precision as it is specified by the parameter `opt` in the routine `dss_create`.

Return Values

- `MKL_DSS_SUCCESS`
- `MKL_DSS_STATE_ERR`
- `MKL_DSS_INVALID_OPTION`
- `MKL_DSS_OPTION_CONFLICT`
- `MKL_DSS_VALUES_ERR`
- `MKL_DSS_OUT_OF_MEMORY`
- `MKL_DSS_ZERO_PIVOT`
- `MKL_DSS_FAILURE`
MKL_DSS_MSG_LVL_ERR
MKL_DSS_TERM_LVL_ERR
MKL_DSS_OOC_MEM_ERR
MKL_DSS_OOC_OC_ERR
MKL_DSS_OOC_RW_ERR

dss_solve_real, dss_solve_complex
Compute the corresponding solution vector and place it in the output array.

Syntax
MKL_INT dss_solve_real(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *rRhsValues, MKL_INT const *nRhs, void *rSolValues)
MKL_INT dss_solve_complex(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, void const *cRhsValues, MKL_INT const *nRhs, void *cSolValues)

Include Files
• mkl.h

Description
For each right-hand side column vector defined in the arrays rRhsValues, cRhsValues, or RhsValues, these routines compute the corresponding solution vector and place it in the arrays rSolValues, cSolValues, or SolValues respectively.

NOTE
The data type (single or double precision) of all arrays must be in correspondence with precision specified by the parameter opt in the routine dss_create.

The lengths of the right-hand side and solution vectors, nRows and nCols respectively, must be defined in a previous call to dss_define_structure.

By default, both routines perform the full solution step (it corresponds to phase = 33 in Intel® MKL PARDISO). The parameter opt enables you to calculate the final solution step-by-step, calling forward and backward substitutions.

If it is set to MKL_DSS_FORWARD_SOLVE, the forward substitution (corresponding to phase = 331 in Intel® MKL PARDISO) is performed;
if it is set to MKL_DSS_DIAGONAL_SOLVE, the diagonal substitution (corresponding to phase = 332 in Intel® MKL PARDISO) is performed, if possible;
if it is set to MKL_DSS_BACKWARD_SOLVE, the backward substitution (corresponding to phase = 333 in Intel® MKL PARDISO) is performed.

For more details about using these substitutions for different types of matrices, see Separate Forward and Backward Substitution in the Intel® MKL PARDISO solver description.

This parameter also can control the number of refinement steps that is used on the solution stage: if it is set to MKL_DSS_REFINEMENT_OFF, the maximum number of refinement steps equal to zero, and if it is set to MKL_DSS_REFINEMENT_ON (default value), the maximum number of refinement steps is equal to 2.
MKL_DSS_CONJUGATE_SOLVE option added to the parameter opt enables solving a conjugate transposed system $A^T X = B$ based on the factorization of the matrix $A$. This option is equivalent to the parameter iparm[11] = 1 in Intel® MKL PARDISO.

MKL_DSS_TRANSPOSE_SOLVE option added to the parameter opt enables solving a transposed system $A^T X = B$ based on the factorization of the matrix $A$. This option is equivalent to the parameter iparm[11] = 2 in Intel® MKL PARDISO.

**Input Parameters**

- **handle**: Pointer to the data structure storing internal DSS results (MKL_DSS_HANDLE).
- **opt**: Parameter to pass the DSS options.
- **nRhs**: Number of the right-hand sides in the system of linear equations.
- **rRhsValues**: Array of size nRows * nRhs. Contains real right-hand side vectors. Real data, single or double precision as it is specified by the parameter opt in the routine dss_create.
- **cRhsValues**: Array of size nRows * nRhs. Contains complex right-hand side vectors. Complex data, single or double precision as it is specified by the parameter opt in the routine dss_create.
- **RhsValues**: Array of size nRows * nRhs. Contains right-hand side vectors. Real or complex data, single or double precision as it is specified by the parameter opt in the routine dss_create.

**Output Parameters**

- **rSolValues**: Array of size nCols * nRhs. Contains real solution vectors. Real data, single or double precision as it is specified by the parameter opt in the routine dss_create.
- **cSolValues**: Array of size nCols * nRhs. Contains complex solution vectors. Complex data, single or double precision as it is specified by the parameter opt in the routine dss_create.

**Return Values**

- MKL_DSS_SUCCESS
- MKL_DSS_STATE_ERR
- MKL_DSS_INVALID_OPTION
- MKL_DSS_OUT_OF_MEMORY
- MKL_DSS_DIAG_ERR
- MKL_DSS_FAILURE
- MKL_DSS_MSG_LVL_ERR
- MKL_DSS_TERM_LVL_ERR
- MKL_DSS_OOC_MEM_ERR
- MKL_DSS_OOC_OC_ERR
**dss_delete**

*Deletes all of the data structures created during the solutions process.*

**Syntax**

```c
MKL_INT dss_delete(_MKL_DSS_HANDLE_t const *handle, MKL_INT const *opt)
```

**Include Files**

- mkl.h

**Description**

The routine `dss_delete` deletes all data structures created during the solving process.

**Input Parameters**

- **opt**
  
  Parameter to pass the DSS options. The default value is `MKL_DSS_MSG_LVL_WARNING + MKL_DSS_TERM_LVL_ERROR`.

**Output Parameters**

- **handle**
  
  Pointer to the data structure storing internal DSS results (`MKL_DSS_HANDLE`).

**Return Values**

- `MKL_DSS_SUCCESS`
- `MKL_DSS_STATE_ERR`
- `MKL_DSS_INVALID_OPTION`
- `MKL_DSS_OUT_OF_MEMORY`
- `MKL_DSS_MSG_LVL_ERR`
- `MKL_DSS_TERM_LVL_ERR`

**dss_statistics**

*Returns statistics about various phases of the solving process.*

**Syntax**

```c
MKL_INT dss_statistics(_MKL_DSS_HANDLE_t *handle, MKL_INT const *opt, _CHARACTER_STR_t const *statArr, _DOUBLE_PRECISION_t *retValues)
```

**Include Files**

- mkl.h

**Description**

The `dss_statistics` routine returns statistics about various phases of the solving process. This routine gathers the following statistics:
- time taken to do reordering,
- time taken to do factorization,
- duration of problem solving,
- determinant of the symmetric indefinite input matrix,
- inertia of the symmetric indefinite input matrix,
- number of floating point operations taken during factorization,
- total peak memory needed during the analysis and symbolic factorization,
- permanent memory needed from the analysis and symbolic factorization,
- memory consumption for the factorization and solve phases.

Statistics are returned in accordance with the input string specified by the parameter `statArr`. The value of the statistics is returned in double precision in a return array, which you must allocate beforehand.

For multiple statistics, multiple string constants separated by commas can be used as input. Return values are put into the return array in the same order as specified in the input string.

Statistics can only be requested at the appropriate stages of the solving process. For example, requesting `FactorTime` before a matrix is factored leads to an error.

The following table shows the point at which each individual statistics item can be requested:

### Statistics Calling Sequences

<table>
<thead>
<tr>
<th>Type of Statistics</th>
<th>When to Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReorderTime</td>
<td>After <code>dss_reorder</code> is completed successfully.</td>
</tr>
<tr>
<td>FactorTime</td>
<td>After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.</td>
</tr>
<tr>
<td>SolveTime</td>
<td>After <code>dss_solve_real</code> or <code>dss_solve_complex</code> is completed successfully.</td>
</tr>
<tr>
<td>Determinant</td>
<td>After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.</td>
</tr>
<tr>
<td>Inertia</td>
<td>After <code>dss_factor_real</code> is completed successfully and the matrix is real, symmetric, and indefinite.</td>
</tr>
<tr>
<td>Flops</td>
<td>After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.</td>
</tr>
<tr>
<td>Peakmem</td>
<td>After <code>dss_reorder</code> is completed successfully.</td>
</tr>
<tr>
<td>Factormem</td>
<td>After <code>dss_reorder</code> is completed successfully.</td>
</tr>
<tr>
<td>Solvemem</td>
<td>After <code>dss_factor_real</code> or <code>dss_factor_complex</code> is completed successfully.</td>
</tr>
</tbody>
</table>

### Input Parameters

- **handle**: Pointer to the data structure storing internal DSS results (`MKL_DSS_HANDLE`).
- **opt**: Parameter to pass the DSS options.
- **statArr**: Input string that defines the type of the returned statistics. The parameter can include one or more of the following string constants (case of the input string has no effect):
  - `ReorderTime`: Amount of time taken to do the reordering.
  - `FactorTime`: Amount of time taken to do the factorization.
  - `SolveTime`: Amount of time taken to solve the problem after factorization.
  - `Determinant`: Determinant of the matrix A.
For real matrices: the determinant is returned as det_pow, det_base in two consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_base}) < 10.0$ and determinant $= \text{det_base} \times 10^{(\text{det_pow})}$.

For complex matrices: the determinant is returned as det_pow, det_re, det_im in three consecutive return array locations, where $1.0 \leq \text{abs}(\text{det_re}) + \text{abs}(\text{det_im}) < 10.0$ and determinant $= (\text{det_re}, \text{det_im}) \times 10^{(\text{det_pow})}$.

**Inertia**

Inertia of a real symmetric matrix is defined as a triplet of nonnegative integers $(p, n, z)$, where $p$ is the number of positive eigenvalues, $n$ is the number of negative eigenvalues, and $z$ is the number of zero eigenvalues.

Inertia is returned as three consecutive return array locations $p, n, z$.

Computing inertia can lead to incorrect results for matrixes with a cluster of eigenvalues which are near 0.

Inertia of a $k$-by-$k$ real symmetric positive definite matrix is always $(k, 0, 0)$. Therefore Inertia is returned only in cases of real symmetric indefinite matrices. For all other matrix types, an error message is returned.

**Flops**

Number of floating point operations performed during the factorization.

**Peakmem**

Total peak memory in kilobytes that the solver needs during the analysis and symbolic factorization phase.

**Factormem**

Permanent memory in kilobytes that the solver needs from the analysis and symbolic factorization phase in the factorization and solve phases.

**Solvemem**

Total double precision memory consumption (kilobytes) of the solver for the factorization and solve phases.

**Output Parameters**

*retValues*  
Value of the statistics returned.

**Finding 'time used to reorder' and 'inertia' of a matrix**

The example below illustrates the use of the dss_statistics routine.
To find the above values, call \texttt{dss\_statistics(handle, opt, statArr, retVal)}, where \texttt{statArr} is "ReorderTime, Inertia"

In this example, \texttt{retVal} has the following values:

\begin{align*}
\texttt{retVal[0]} & \quad \text{Time to reorder.} \\
\texttt{retVal[1]} & \quad \text{Positive Eigenvalues.} \\
\texttt{retVal[2]} & \quad \text{Negative Eigenvalues.} \\
\texttt{retVal[3]} & \quad \text{Zero Eigenvalues.}
\end{align*}

\textbf{Return Values}

\begin{itemize}
\item \texttt{MKL\_DSS\_SUCCESS}
\item \texttt{MKL\_DSS\_INVALID\_OPTION}
\item \texttt{MKL\_DSS\_STATISTICS\_INVALID\_MATRIX}
\item \texttt{MKL\_DSS\_STATISTICS\_INVALID\_STATE}
\item \texttt{MKL\_DSS\_STATISTICS\_INVALID\_STRING}
\item \texttt{MKL\_DSS\_MSG\_LVL\_ERR}
\item \texttt{MKL\_DSS\_TERM\_LVL\_ERR}
\end{itemize}

\section*{Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS)}

Intel\textsuperscript{®} MKL supports iterative sparse solvers (ISS) based on the reverse communication interface (RCI), referred to here as the RCI ISS interface. The RCI ISS interface implements a group of user-callable routines that are used in the step-by-step solving process of a symmetric positive definite system (RCI conjugate gradient solver, or RCI CG), and of a non-symmetric indefinite (non-degenerate) system (RCI flexible generalized minimal residual solver, or RCI FGMRES) of linear algebraic equations. This interface uses the general RCI scheme described in [Dong95].

See the Appendix A Linear Solvers Basics for discussion of terms and concepts related to the ISS routines.

The term \textit{RCI} indicates that when the solver needs the results of certain operations (for example, matrix-vector multiplications), the user performs them and passes the result to the solver. This makes the solver more universal as it is independent of the specific implementation of the operations like the matrix-vector multiplication. To perform such operations, the user can use the built-in sparse matrix-vector multiplications and triangular solvers routines described in Sparse BLAS Level 2 and Level 3 Routines.

\textbf{NOTE}

The RCI CG solver is implemented in two versions: for system of equations with a single right-hand side, and for systems of equations with multiple right-hand sides.

The CG method may fail to compute the solution or compute the wrong solution if the matrix of the system is not symmetric and not positive definite.

The FGMRES method may fail if the matrix is degenerate.

Table "RCI CG Interface Routines" lists the names of the routines, and describes their general use.
### RCI ISS Interface Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcg_init, dcmrhs_init, dfgmres_init</td>
<td>Initializes the solver.</td>
</tr>
<tr>
<td>dcg_check, dcmrhs_check, dfgmres_check</td>
<td>Checks the consistency and correctness of the user defined data.</td>
</tr>
<tr>
<td>dcg, dcmrhs, dfgmres</td>
<td>Computes the approximate solution vector.</td>
</tr>
<tr>
<td>dcm_get, dcmrhs_get, dfgmres_get</td>
<td>Retrieves the number of the current iteration.</td>
</tr>
</tbody>
</table>

The Intel® MKL RCI ISS interface routines are normally invoked in this order:

1. `<system_type>_init`
2. `<system_type>_check`
3. `<system_type>`
4. `<system_type>_get`

Advanced users can change that order if they need it. Others should follow the above order of calls.

The following diagram indicates the typical order in which the RCI ISS interface routines are invoked:

__border__top

**Typical Order for Invoking RCI ISS interface Routines**

---

See the code examples that use the RCI ISS interface routines to solve systems of linear equations in the Intel® MKL installation directory.

- examples/solverc/source

---

### CG Interface Description

Each routine for the RCI CG solver is implemented in two versions: for a system of equations with a single right-hand side (SRHS), and for a system of equations with multiple right-hand sides (MRHS). The names of routines for a system with MRHS contain the suffix `mrhs`.

### Routine Options

All of the RCI CG routines have common parameters for passing various options to the routines (see CG Common Parameters). The values for these parameters can be changed during computations.
**User Data Arrays**

Many of the RCI CG routines take arrays of user data as input. For example, user arrays are passed to the routine `dcg` to compute the solution of a system of linear algebraic equations. The Intel® MKL RCI CG routines do not make copies of the user input arrays to minimize storage requirements and improve overall run-time efficiency.

**CG Common Parameters**

NOTE
The default and initial values listed below are assigned to the parameters by calling the `dcg_init/dcgmrhs_init` routine.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td>MKL_INT, this parameter sets the size of the problem in the <code>dcg_init/dcgmrhs_init</code> routine. All the other routines use the <code>ipar[0]</code> parameter instead. Note that the coefficient matrix <code>A</code> is a square matrix of size <code>n*n</code>.</td>
</tr>
<tr>
<td><code>x</code></td>
<td>double array of size <code>n</code> for SRHS, or matrix of size <code>(n*nrhs)</code> for MRHS. This parameter contains the current approximation to the solution. Before the first call to the <code>dcg/dcgmrhs</code> routine, it contains the initial approximation to the solution.</td>
</tr>
<tr>
<td><code>nrhs</code></td>
<td>MKL_INT, this parameter sets the number of right-hand sides for MRHS routines.</td>
</tr>
<tr>
<td><code>b</code></td>
<td>double array containing a single right-hand side vector, or matrix of size <code>n*nrhs</code> containing right-hand side vectors.</td>
</tr>
<tr>
<td><code>RCI_request</code></td>
<td>MKL_INT, this parameter gives information about the result of work of the RCI CG routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:</td>
</tr>
</tbody>
</table>

- **RCI_request= 1**
  multiply the matrix by `tmp[0:n-1]`, put the result in `tmp[n:2*n-1]`, and return the control to the `dcg/dcgmrhs` routine;

- **RCI_request= 2**
  to perform the stopping tests. If they fail, return the control to the `dcg/dcgmrhs` routine. If the stopping tests succeed, it indicates that the solution is found and stored in the `x` array;

- **RCI_request= 3**
  for SRHS: apply the preconditioner to `tmp[2*n:3*n-1]`, put the result in `tmp[3*n:4*n-1]`, and return the control to the `dcg` routine;
  for MRHS: apply the preconditioner to `tmp[2+ipar[2]*n:(3 + ipar[2])*n - 1]`, put the result in `tmp[3*n:4*n - 1]`, and return the control to the `dcgmrhs` routine.

Note that the `dcg_get/dcgmrhs_get` routine does not change the parameter `RCI_request`. This enables use of this routine inside the reverse communication computations.
MKL_INT array, of size 128 for SRHS, and of size \((128+2*\text{nrhs})\) for MRHS. This parameter specifies the integer set of data for the RCI CG computations:

- **ipar[0]** specifies the size of the problem. The dcg_init/dcgmrhs_init routine assigns \(\text{ipar}[0]=n\). All the other routines use this parameter instead of \(n\). There is no default value for this parameter.

- **ipar[1]** specifies the type of output for error and warning messages generated by the RCI CG routines. The default value 6 means that all messages are displayed on the screen. Otherwise, the error and warning messages are written to the newly created files dcg_errors.txt and dcg_check_warnings.txt, respectively. Note that if \(\text{ipar}[5]\) and \(\text{ipar}[6]\) parameters are set to 0, error and warning messages are not generated at all.

- **ipar[2]** for SRHS: contains the current stage of the RCI CG computations. The initial value is 1;
  for MRHS: contains the number of the right-hand side for which the calculations are currently performed.

  **WARNING** Avoid altering this variable during computations.

- **ipar[3]** contains the current iteration number. The initial value is 0.

- **ipar[4]** specifies the maximum number of iterations. The default value is \(\min(150, n)\).

- **ipar[5]** if the value is not equal to 0, the routines output error messages in accordance with the parameter \(\text{ipar}[1]\). Otherwise, the routines do not output error messages at all, but return a negative value of the parameter \(\text{RCI\_request}\). The default value is 1.

- **ipar[6]** if the value is not equal to 0, the routines output warning messages in accordance with the parameter \(\text{ipar}[1]\). Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter \(\text{RCI\_request}\). The default value is 1.

- **ipar[7]** if the value is not equal to 0, the dcg/dcgmrhs routine performs the stopping test for the maximum number of iterations: \(\text{ipar}[3] \leq\)
Otherwise, the method is stopped and the corresponding value is assigned to the \textit{RCI\_request}. If the value is 0, the routine does not perform this stopping test. The default value is 1.

\textbf{ipar[8]} \begin{itemize}
  \item if the value is not equal to 0, the \texttt{dcg/dcgmrhs} routine performs the residual stopping test:
    \begin{equation}
      \text{dpar}(5) \leq \text{dpar}(4) = \text{dpar}(1) \times \text{dpar}(3) + \text{dpar}(2).
    \end{equation}
  \item Otherwise, the method is stopped and the corresponding value is assigned to the \textit{RCI\_request}. If the value is 0, the routine does not perform this stopping test. The default value is 0.
\end{itemize}

\textbf{ipar[9]} \begin{itemize}
  \item if the value is not equal to 0, the \texttt{dcg/dcgmrhs} routine requests a user-defined stopping test by setting the output parameter \textit{RCI\_request}=2.
  \item If the value is 0, the routine does not perform the user-defined stopping test. The default value is 1.
\end{itemize}

\textbf{NOTE} At least one of the parameters \texttt{ipar[7]}-\texttt{ipar[9]} must be set to 1.

\textbf{ipar[10]} \begin{itemize}
  \item if the value is equal to 0, the \texttt{dcg/dcgmrhs} routine runs the non-preconditioned version of the corresponding CG method. Otherwise, the routine runs the preconditioned version of the CG method, and by setting the output parameter \textit{RCI\_request}=3, indicates that you must perform the preconditioning step. The default value is 0.
\end{itemize}

\textbf{ipar[11:127]} are reserved and not used in the current RCI CG SRHS and MRHS routines.

\textbf{NOTE} For future compatibility, you must declare the array \texttt{ipar} with length 128 for a single right-hand side.

\textbf{ipar[11:127]} + \texttt{2*nrhs]} \begin{itemize}
  \item are reserved for internal use in the current RCI CG SRHS and MRHS routines.
\end{itemize}

\textbf{NOTE} For future compatibility, you must declare the array \texttt{ipar} with length 128+2\texttt{*nrhs} for multiple right-hand sides.
**dpar**

*double array, for SRHS of size 128, for MRHS of size \((128+2*\text{nrhs})\); this parameter is used to specify the double precision set of data for the RCI CG computations, specifically:*

- **dpar[0]** specifies the relative tolerance. The default value is \(1.0 \times 10^{-6}\).
- **dpar[1]** specifies the absolute tolerance. The default value is 0.0.
- **dpar[2]** specifies the square norm of the initial residual (if it is computed in the \text{dgc/dcgmrhs} routine). The initial value is 0.0.
- **dpar[3]** service variable equal to \(dpar[0] * dpar[2] + dpar[1]\) (if it is computed in the \text{dgc/dcgmrhs} routine). The initial value is 0.0.
- **dpar[4]** specifies the square norm of the current residual. The initial value is 0.0.
- **dpar[5]** specifies the square norm of residual from the previous iteration step (if available). The initial value is 0.0.
- **dpar[6]** contains the \textit{alpha} parameter of the CG method. The initial value is 0.0.
- **dpar[7]** contains the \textit{beta} parameter of the CG method, it is equal to \(dpar[4]/dpar[5]\) The initial value is 0.0.
- **dpar[8:127]** are reserved and not used in the current RCI CG SRHS and MRHS routines.

**NOTE**

For future compatibility, you must declare the array \textit{dpar} with length 128 for a single right-hand side.

- **dpar(9:128+2*\text{nrhs})** [8:127 + 2*\text{nrhs}] are reserved for internal use in the current RCI CG SRHS and MRHS routines.

**NOTE**

For future compatibility, you must declare the array \textit{dpar} with length \(128+2*\text{nrhs}\) for multiple right-hand sides.

**tmp**

*double array of size \((n*4)\) for SRHS, and \((n*(3+\text{nrhs}))\) for MRHS. This parameter is used to supply the double precision temporary space for the RCI CG computations, specifically:*
tmp[0:n - 1] specifies the current search direction. The initial value is 0.0.

tmp[n:2*n - 1] contains the matrix multiplied by the current search direction. The initial value is 0.0.

tmp[2*n:3*n - 1] contains the current residual. The initial value is 0.0.

tmp[3*n:4*n - 1] contains the inverse of the preconditioner applied to the current residual for the SRHS version of CG. There is no initial value for this parameter.

tmp[4*n:(4 + nrhs)*n - 1] contains the inverse of the preconditioner applied to the current residual for the MRHS version of CG. There is no initial value for this parameter.

NOTE
You can define this array in the code using RCI CG SRHS as double tmp[3*n] if you run only non-preconditioned CG iterations.

FGMRES Interface Description

Routine Options
All of the RCI FGMRES routines have common parameters for passing various options to the routines (see FGMRES Common Parameters). The values for these parameters can be changed during computations.

User Data Arrays
Many of the RCI FGMRES routines take arrays of user data as input. For example, user arrays are passed to the routine dfgmres to compute the solution of a system of linear algebraic equations. To minimize storage requirements and improve overall run-time efficiency, the Intel® MKL RCI FGMRES routines do not make copies of the user input arrays.

FGMRES Common Parameters

NOTE
The default and initial values listed below are assigned to the parameters by calling the dfgmres_init routine.

n MKL_INT, this parameter sets the size of the problem in the dfgmres_init routine. All the other routines use the ipar[0] parameter instead. Note that the coefficient matrix A is a square matrix of size n*n.

x double array, this parameter contains the current approximation to the solution vector. Before the first call to the dfgmres routine, it contains the initial approximation to the solution vector.
double array, this parameter contains the right-hand side vector. Depending on user requests (see the parameter ipar[12]), it might contain the approximate solution after execution.

RCI_request

MKL_INT, this parameter gives information about the result of work of the RCI FGMRES routines. Negative values of the parameter indicate that the routine completed with errors or warnings. The 0 value indicates successful completion of the task. Positive values mean that you must perform specific actions:

- **RCI_request= 1**
  - multiply the matrix by \( \text{tmp[ipar[21]} - 1:ipar[21] + n - 2] \), put the result in \( \text{tmp[ipar[22]} - 1:ipar[22] + n - 2] \), and return the control to the dfgmres routine;

- **RCI_request= 2**
  - perform the stopping tests. If they fail, return the control to the dfgres routine. Otherwise, the solution can be updated by a subsequent call to dfgmres_get routine;

- **RCI_request= 3**

- **RCI_request= 4**
  - check if the norm of the current orthogonal vector is zero, within the rounding or computational errors. Return the control to the dfgmres routine if it is not zero, otherwise complete the solution process by calling dfgmres_get routine.

ipar[128]

MKL_INT array, this parameter specifies the integer set of data for the RCI FGMRES computations:

- **ipar[0]** specifies the size of the problem. The dfgmres_init routine assigns \( ipar[0]=n \). All the other routines uses this parameter instead of \( n \). There is no default value for this parameter.

- **ipar[1]** specifies the type of output for error and warning messages that are generated by the RCI FGMRES routines. The default value 6 means that all messages are displayed on the screen. Otherwise the error and warning messages are written to the newly created file MKL_RCI_FGMRES_Log.txt. Note that if \( ipar[5] \) and \( ipar[6] \) parameters are set to 0, error and warning messages are not generated at all.

- **ipar[2]** contains the current stage of the RCI FGMRES computations. The initial value is 1.
**WARNING**
Avoid altering this variable during computations.

- **ipar[3]** contains the current iteration number. The initial value is 0.
- **ipar[4]** specifies the maximum number of iterations. The default value is min (150, n).
- **ipar[5]** if the value is not 0, the routines output error messages in accordance with the parameter ipar[1]. If it is 0, the routines do not output error messages at all, but return a negative value of the parameter RCI_request. The default value is 1.
- **ipar[6]** if the value is not 0, the routines output warning messages in accordance with the parameter ipar[1]. Otherwise, the routines do not output warning messages at all, but they return a negative value of the parameter RCI_request. The default value is 1.
- **ipar[7]** if the value is not equal to 0, the dfgmres routine performs the stopping test for the maximum number of iterations: ipar[3] ≤ ipar[4]. If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 1.
- **ipar[8]** if the value is not 0, the dfgmres routine performs the residual stopping test: dpar[4] ≤ dpar[3]. If the value is 0, the dfgmres routine does not perform this stopping test. The default value is 0.
- **ipar[9]** if the value is not 0, the dfgmres routine indicates that the user-defined stopping test should be performed by setting RCI_request=2. If the value is 0, the dfgmres routine does not perform the user-defined stopping test. The default value is 1.

**NOTE**
At least one of the parameters ipar[7]-ipar[9] must be set to 1.

- **ipar[10]** if the value is 0, the dfgmres routine runs the non-preconditioned version of the FGMRES method. Otherwise, the routine runs the preconditioned version of the FGMRES method,
and requests that you perform the preconditioning step by setting the output parameter \( RCI\_request=3 \). The default value is 0.

If the value is not equal to 0, the \( dfgmres \) routine performs the automatic test for zero norm of the currently generated vector:
\[ dpar[6] \leq dpar[7], \]
where \( dpar[7] \) contains the tolerance value. Otherwise, the routine indicates that you must perform this check by setting the output parameter \( RCI\_request=4 \). The default value is 0.

If the value is equal to 0, the \( dfgmres\_get \) routine updates the solution to the vector \( x \) according to the computations done by the \( dfgmres \) routine. If the value is positive, the routine writes the solution to the right-hand side vector \( b \). If the value is negative, the routine returns only the number of the current iteration, and does not update the solution. The default value is 0.

**NOTE**
It is possible to call the \( dfgmres\_get \) routine at any place in the code, but you must pay special attention to the parameter \( ipar[12] \). The RCI FGMRES iterations can be continued after the call to \( dfgmres\_get \) routine only if the parameter \( ipar[12] \) is not equal to zero. If \( ipar[12] \) is positive, then the updated solution overwrites the right-hand side in the vector \( b \). If you want to run the restarted version of FGMRES with the same right-hand side, then it must be saved in a different memory location before the first call to the \( dfgmres\_get \) routine with positive \( ipar[12] \).

The internal iteration counter that counts the number of iterations before the restart takes place. The initial value is 0.

**WARNING**
Do not alter this variable during computations.

Specifies the number of the non-restarted FGMRES iterations. To run the restarted version of the FGMRES method, assign the number of
iterations to \( \text{ipar}[14] \) before the restart. The default value is \( \min(150, n) \), which means that by default the non-restarted version of FGMRES method is used.

\( \text{ipar}[15] \) service variable specifying the location of the rotated Hessenberg matrix from which the matrix stored in the packed format (see Matrix Arguments in the Appendix B for details) is started in the \( \text{tmp} \) array.

\( \text{ipar}[16] \) service variable specifying the location of the rotation cosines from which the vector of cosines is started in the \( \text{tmp} \) array.

\( \text{ipar}[17] \) service variable specifying the location of the rotation sines from which the vector of sines is started in the \( \text{tmp} \) array.

\( \text{ipar}[18] \) service variable specifying the location of the rotated residual vector from which the vector is started in the \( \text{tmp} \) array.

\( \text{ipar}[19] \) service variable, specifies the location of the least squares solution vector from which the vector is started in the \( \text{tmp} \) array.

\( \text{ipar}[20] \) service variable specifying the location of the set of preconditioned vectors from which the set is started in the \( \text{tmp} \) array. The memory locations in the \( \text{tmp} \) array starting from \( \text{ipar}[20] \) are used only for the preconditioned FGMRES method.

\( \text{ipar}[21] \) specifies the memory location from which the first vector (source) used in operations requested via \( \text{RCI\_request} \) is started in the \( \text{tmp} \) array.

\( \text{ipar}[22] \) specifies the memory location from which the second vector (output) used in operations requested via \( \text{RCI\_request} \) is started in the \( \text{tmp} \) array.

\( \text{ipar}[23:127] \) are reserved and not used in the current RCI FGMRES routines.

**NOTE**
You must declare the array \( \text{ipar} \) with length 128. While defining the array in the code as \( \text{ipar}[23] \) works, there is no guarantee of future compatibility with Intel® MKL.

\( \text{dpar}(128) \) double array, this parameter specifies the double precision set of data for the RCI CG computations, specifically:
dpar[0] specifies the relative tolerance. The default value is 1.0e-6.
dpar[1] specifies the absolute tolerance. The default value is 0.0e-0.
dpar[2] specifies the Euclidean norm of the initial residual (if it is computed in the dfgmres routine). The initial value is 0.0.
dpar[3] service variable equal to dpar[0]*dpar[2]+dpar[1] (if it is computed in the dfgmres routine). The initial value is 0.0.
dpar[4] specifies the Euclidean norm of the current residual. The initial value is 0.0.
dpar[5] specifies the Euclidean norm of residual from the previous iteration step (if available). The initial value is 0.0.
dpar[6] contains the norm of the generated vector. The initial value is 0.0.

**NOTE**
In terms of [Saad03] this parameter is the coefficient \(h_{k+1,k}\) of the Hessenberg matrix.

dpar[7] contains the tolerance for the zero norm of the currently generated vector. The default value is 1.0e-12.
dpar[8:127] are reserved and not used in the current RCI FGMRES routines.

**NOTE**
You must declare the array \(dpar\) with length 128. While defining the array in the code as double \(dpar[8]\) works, there is no guarantee of future compatibility with Intel® MKL.

double array of size \(((2\times ipar[14] + 1)\times n + ipar[14]\times(ipar[14] + 9)/2 + 1))/2\) used to supply the double precision temporary space for the RCI FGMRES computations, specifically:

tmp[0:ipar[15] - 2] contains the sequence of vectors generated by the FGMRES method. The initial value is 0.0.
tmp[ipar[15] - 1:ipar[16] - 2] contains the rotated Hessenberg matrix generated by the FGMRES method; the matrix is stored in the packed format. There is no initial value for this part of \(tmp\) array.


tmp[ipar[20] - 1:*] contains the set of preconditioned vectors generated for the FGMRES method by the user. This part of tmp array is not used if the non-preconditioned version of FGMRES method is called. There is no initial value for this part of tmp array.

**NOTE**
You can define this array in the code as double tmp[(2*ipar[14] + 1)*n + ipar[14]*(ipar[14] + 9)/2 + 1)] if you run only non-preconditioned FGMRES iterations.

---

**RCI ISS Routines**

**dcg_init**

*Initializes the solver.*

**Syntax**

```c
void dcg_init (const MKL_INT *n, const double *x, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp );
```

**Include Files**

- mkl.h

**Description**

The routine `dcg_init` initializes the solver. After initialization, all subsequent invocations of the Intel® MKL RCI CG routines use the values of all parameters returned by the routine `dcg_init`. Advanced users can skip this step and set the values in the `ipar` and `dpar` arrays directly.
Caution
You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the \texttt{dcg\_check} routine, but it does not guarantee that the method will work correctly.

Input Parameters

\begin{tabular}{ll}
\textit{n} & Sets the size of the problem. \\
\textit{x} & Array of size \textit{n}. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to \textit{b}. \\
\textit{b} & Array of size \textit{n}. Contains the right-hand side vector.
\end{tabular}

Output Parameters

\begin{tabular}{ll}
\textit{RCI\_request} & Gives information about the result of the routine. \\
\textit{ipar} & Array of size 128. Refer to the \textit{CG Common Parameters}. \\
\textit{dpar} & Array of size 128. Refer to the \textit{CG Common Parameters}. \\
\textit{tmp} & Array of size \((n*4)\). Refer to the \textit{CG Common Parameters}.
\end{tabular}

Return Values

\begin{tabular}{ll}
\textit{RCI\_request} = 0 & Indicates that the task completed normally. \\
\textit{RCI\_request} = -10000 & Indicates failure to complete the task.
\end{tabular}

dcg\_check

\textit{Checks consistency and correctness of the user defined data.}

Syntax

\begin{verbatim}
void dcg_check (const MKL_INT *n, const double *x, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
\end{verbatim}

Include Files

\begin{itemize}
\item mkl.h
\end{itemize}

Description

The routine \texttt{dcg\_check} checks consistency and correctness of the parameters to be passed to the solver routine \texttt{dcg}. However this operation does not guarantee that the solver returns the correct result. It only reduces the chance of making a mistake in the parameters of the method. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors must be defined in a previous call to the \texttt{dcg\_init} routine.

Input Parameters

\begin{tabular}{ll}
\textit{n} & Sets the size of the problem.
\end{tabular}
Array of size $n$. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to $b$.

Array of size $n$. Contains the right-hand side vector.

Output Parameters

$RCI_{\text{request}}$ Gives information about result of the routine.

Array of size 128. Refer to the CG Common Parameters.

Array of size 128. Refer to the CG Common Parameters.

Array of size $(n*4)$. Refer to the CG Common Parameters.

Return Values

$RCI_{\text{request}} = 0$ Indicates that the task completed normally.

$RCI_{\text{request}} = -1100$ Indicates that the task is interrupted and the errors occur.

$RCI_{\text{request}} = -1001$ Indicates that there are some warning messages.

$RCI_{\text{request}} = -1010$ Indicates that the routine changed some parameters to make them consistent or correct.

$RCI_{\text{request}} = -1011$ Indicates that there are some warning messages and that the routine changed some parameters.

dcg

Computes the approximate solution vector.

Syntax

```c
void dcg (const MKL_INT *n , double *x , const double *b , MKL_INT *RCI_request , MKL_INT *ipar , double *dpar , double *tmp );
```

Include Files

- mkl.h

Description

The dcg routine computes the approximate solution vector using the CG method [Young71]. The routine dcg uses the vector in the array $x$ before the first call as an initial approximation to the solution. The parameter $RCI_{\text{request}}$ gives you information about the task completion and requests results of certain operations that are required by the solver.

Note that lengths of all vectors must be defined in a previous call to the dcg_init routine.

Input Parameters

$n$ Sets the size of the problem.

Array of size $n$. Contains the initial approximation to the solution vector.

Array of size $n$. Contains the right-hand side vector.
tmp

Array of size \((n*4)\). Refer to the CG Common Parameters.

**Output Parameters**

*RCI_request*

Gives information about result of work of the routine.

*x*

Array of size \(n\). Contains the updated approximation to the solution vector.

*ipar*

Array of size 128. Refer to the CG Common Parameters.

*dpar*

Array of size 128. Refer to the CG Common Parameters.

*tmp*

Array of size \((n*4)\). Refer to the CG Common Parameters.

**Return Values**

*RCI_request=0*

Indicates that the task completed normally and the solution is found and stored in the vector \(x\). This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the *RCI_request= 2*.

*RCI_request=-1*

Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.

*RCI_request=-2*

Indicates that the routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.

*RCI_request=- 10*

Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value \(dpar[5]\) was altered outside of the routine, or the \(dcg\_check\) routine was not called.

*RCI_request=-11*

Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values \(ipar[7], ipar[8], ipar[9]\) were altered outside of the routine, or the \(dcg\_check\) routine was not called.

*RCI_request= 1*

Indicates that you must multiply the matrix by \(tmp[0:n - 1]\), put the result in the \(tmp[n:2*n - 1]\), and return control back to the routine \(dcg\).

*RCI_request= 2*

Indicates that you must perform the stopping tests. If they fail, return control back to the \(dcg\) routine. Otherwise, the solution is found and stored in the vector \(x\).

*RCI_request= 3*

Indicates that you must apply the preconditioner to \([2*n: 3*n - 1]\), put the result in the \([3*n:4*n - 1]\), and return control back to the routine \(dcg\).

**dcg_get**

Retrieves the number of the current iteration.
Syntax

void dcg_get(const MKL_INT *n, const double *x, const double *b, const MKL_INT *RCI_request, const MKL_INT *ipar, const double *dpar, const double *tmp, MKL_INT *itercount);

Include Files

• mkl.h

Description

The routine dcg_get retrieves the current iteration number of the solutions process.

Input Parameters

- \(n\): Sets the size of the problem.
- \(x\): Array of size \(n\). Contains the approximation vector to the solution.
- \(b\): Array of size \(n\). Contains the right-hand side vector.
- \(RCI\_request\): This parameter is not used.
- \(ipar\): Array of size 128. Refer to the CG Common Parameters.
- \(dpar\): Array of size 128. Refer to the CG Common Parameters.
- \(tmp\): Array of size \((n*4)\). Refer to the CG Common Parameters.

Output Parameters

- \(itercount\): Returns the current iteration number.

Return Values

The routine dcg_get has no return values.

dcgmrhs_init

Initializes the RCI CG solver with MHRS.

Syntax

void dcgmrhs_init(const MKL_INT *n, const double *x, const MKL_INT *nrhs, const double *b, const MKL_INT *method, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);

Include Files

• mkl.h

Description

The routine dcgmrhs_init initializes the solver. After initialization all subsequent invocations of the Intel® MKL RCI CG with multiple right-hand sides (MRHS) routines use the values of all parameters that are returned by dcgmrhs_init. Advanced users may skip this step and set the values to these parameters directly in the appropriate routines.
WARNING
You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the dcgmrhs_check routine, but it does not guarantee that the method will work correctly.

Input Parameters

- \( n \)  
  Sets the size of the problem.

- \( x \)  
  Array of size \( n \times nrhs \). Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to \( b \).

- \( nrhs \)  
  Sets the number of right-hand sides.

- \( b \)  
  Array of size \( n \times nrhs \). Contains the right-hand side vectors.

- method  
  Specifies the method of solution: 
  A value of 1 indicates CG with multiple right-hand sides (default value)

Output Parameters

- RCI_request  
  Gives information about the result of the routine.

  - ipar  
    Array of size \( (128+2 \times nrhs) \). Refer to the CG Common Parameters.

  - dpar  
    Array of size \( (128+2 \times nrhs) \). Refer to the CG Common Parameters.

  - tmp  
    Array of size \( (n \times (3+nrhs)) \). Refer to the CG Common Parameters.

Return Values

- \( RCI\_request = 0 \)  
  Indicates that the task completed normally.

- \( RCI\_request = -10000 \)  
  Indicates failure to complete the task.

dcgmrhs_check

Checks consistency and correctness of the user defined data.

Syntax

```c
void dcgmrhs_check (const MKL_INT *n, const double *x, const MKL_INT *nrhs, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- mkl.h

Description

The routine dcgmrhs_check checks the consistency and correctness of the parameters to be passed to the solver routine dcgmrhs. While this operation reduces the chance of making a mistake in the parameters, it does not guarantee that the solver returns the correct result.

If you are sure that the correct data is specified in the solver parameters, you can skip this operation.

The lengths of all vectors must be defined in a previous call to the dcgmrhs_init routine.
Input Parameters

\( n \)  
Sets the size of the problem.

\( x \)  
Array of size \( n \times nrhs \). Contains the initial approximation to the solution vectors. Normally it is equal to 0 or to \( b \).

\( nrhs \)  
This parameter sets the number of right-hand sides.

\( b \)  
Array of size \( n \times nrhs \). Contains the right-hand side vectors.

Output Parameters

\( RCI\_request \)  
Returns information about the results of the routine.

\( ipar \)  
Array of size \((128+2 \times nrhs)\). Refer to the CG Common Parameters.

\( dpar \)  
Array of size \((128+2 \times nrhs)\). Refer to the CG Common Parameters.

\( tmp \)  
Array of size \((n \times (3+nrhs))\). Refer to the CG Common Parameters.

Return Values

\( RCI\_request = 0 \)  
Indicates that the task completed normally.

\( RCI\_request = -1100 \)  
Indicates that the task is interrupted and the errors occur.

\( RCI\_request = -1001 \)  
Indicates that there are some warning messages.

\( RCI\_request = -1010 \)  
Indicates that the routine changed some parameters to make them consistent or correct.

\( RCI\_request = -1011 \)  
Indicates that there are some warning messages and that the routine changed some parameters.

dcgmrhs

*Computes the approximate solution vectors.*

Syntax

```c
void dcgmrhs (const MKL_INT *n, double *x, const MKL_INT *nrhs, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

Include Files

- mkl.h

Description

The routine \texttt{dcgmrhs} computes approximate solution vectors using the CG with multiple right-hand sides (MRHS) method [Young71]. The routine \texttt{dcgmrhs} uses the value that was in the \( x \) before the first call as an initial approximation to the solution. The parameter \( RCI\_request \) gives information about task completion status and requests results of certain operations that are required by the solver.

Note that lengths of all vectors are assumed to have been defined in a previous call to the \texttt{dcgmrhs\_init} routine.
**Input Parameters**

- **n**
  
  Sets the size of the problem, and the sizes of arrays \( x \) and \( b \).

- **x**
  
  Array of size \( n \times nrhs \). Contains the initial approximation to the solution vectors.

- **nrhs**
  
  Sets the number of right-hand sides.

- **b**
  
  Array of size \( n \times nrhs \). Contains the right-hand side vectors.

- **tmp**
  
  Array of size \((n, 3+nrhs)\). Refer to the CG Common Parameters.

**Output Parameters**

- **RCI_request**
  
  Gives information about result of work of the routine.

- **x**
  
  Array of size \((n \times nrhs)\). Contains the updated approximation to the solution vectors.

- **ipar**
  
  Array of size \((128 + 2 \times nrhs)\). Refer to the CG Common Parameters.

- **dpar**
  
  Array of size \((128 + 2 \times nrhs)\). Refer to the CG Common Parameters.

- **tmp**
  
  Array of size \((n \times (3+nrhs))\). Refer to the CG Common Parameters.

**Return Values**

- **RCI_request = 0**
  
  Indicates that the task completed normally and the solution is found and stored in the vector \( x \). This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the \( RCI\_request = 2 \).

- **RCI_request = -1**
  
  Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if both tests are requested by the user.

- **RCI_request = -2**
  
  The routine was interrupted because of an attempt to divide by zero. This situation happens if the matrix is non-positive definite or almost non-positive definite.

- **RCI_request = -10**
  
  Indicates that the routine was interrupted because the residual norm is invalid. This usually happens because the value \( dpar[5] \) was altered outside of the routine, or the \( dcg\_check \) routine was not called.

- **RCI_request = -11**
  
  Indicates that the routine was interrupted because it enters the infinite cycle. This usually happens because the values \( ipar[7], ipar[8], ipar[9] \) were altered outside of the routine, or the \( dcg\_check \) routine was not called.

- **RCI_request = 1**
  
  Indicates that you must multiply the matrix by \( tmp[0:n - 1] \), put the result in the \( tmp[n:2n - 1] \), and return control back to the routine \( dcg \).
$RCI_{request}= 2$

Indicates that you must perform the stopping tests. If they fail, return control back to the $dcg$ routine. Otherwise, the solution is found and stored in the vector $x$.

$RCI_{request}= 3$

Indicates that you must apply the preconditioner to $tmp[2*n:3*n - 1]$, put the result in the $tmp[3*n:4*n - 1]$, and return control back to the routine $dcg$.

dcgmrhs_get

Retrieves the number of the current iteration.

Syntax

```c
void dcgmrhs_get (const MKL_INT *n, const double *x, const MKL_INT *nrhs, const double *b, const MKL_INT *RCI_request, const MKL_INT *ipar, const double *dpar, const double *tmp, MKL_INT *itercount);
```

Include Files

- mkl.h

Description

The routine $dcgmrhs_get$ retrieves the current iteration number of the solving process.

Input Parameters

$n$

Sets the size of the problem.

$x$

Array of size $n*nrhs$. Contains the initial approximation to the solution vectors.

$nrhs$

Sets the number of right-hand sides.

$b$

Array of size $n*nrhs$. Contains the right-hand side.

RCI_request

This parameter is not used.

ipar

Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters.

dpar

Array of size $(128+2*nrhs)$. Refer to the CG Common Parameters.

tmp

Array of size $(n*(3+nrhs))$. Refer to the CG Common Parameters.

Output Parameters

itercount

Array of size $nrhs$. Returns the current iteration number for each right-hand side.

Return Values

The routine $dcgmrhs_get$ has no return values.

dfgmres_init

Initializes the solver.
### Syntax

```c
void dfgmres_init (const MKL_INT *n, const double *x, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```

### Include Files
- mkl.h

### Description

The routine `dfgmres_init` initializes the solver. After initialization all subsequent invocations of Intel® MKL RCI FGMRES routines use the values of all parameters that are returned by `dfgmres_init`. Advanced users can skip this step and set the values in the `ipar` and `dpar` arrays directly.

---

**WARNING**

You can modify the contents of these arrays after they are passed to the solver routine only if you are sure that the values are correct and consistent. You can perform a basic check for correctness and consistency by calling the `dfgmres_check` routine, but it does not guarantee that the method will work correctly.

---

### Input Parameters

- **n**
  - Sets the size of the problem.

- **x**
  - Array of size `n`. Contains the initial approximation to the solution vector. Normally it is equal to 0 or to `b`.

- **b**
  - Array of size `n`. Contains the right-hand side vector.

### Output Parameters

- **RCI_request**
  - Gives information about the result of the routine.

- **ipar**
  - Array of size 128. Refer to the FGMRES Common Parameters.

- **dpar**
  - Array of size 128. Refer to the FGMRES Common Parameters.

- **tmp**

### Return Values

- **RCI_request= 0**
  - Indicates that the task completed normally.

- **RCI_request= -10000**
  - Indicates failure to complete the task.

### dfgmres_check

*Checks consistency and correctness of the user defined data.*

#### Syntax

```c
void dfgmres_check (const MKL_INT *n, const double *x, const double *b, MKL_INT *RCI_request, MKL_INT *ipar, double *dpar, double *tmp);
```
Include Files
• mkl.h

Description
The routine dfgmres_check checks consistency and correctness of the parameters to be passed to the solver routine dfgmres. However, this operation does not guarantee that the method gives the correct result. It only reduces the chance of making a mistake in the parameters of the routine. Skip this operation only if you are sure that the correct data is specified in the solver parameters.

The lengths of all vectors are assumed to have been defined in a previous call to the dfgmres_init routine.

Input Parameters

\( n \)

Sets the size of the problem.

\( x \)

Array of size \( n \). Contains the initial approximation to the solution vector. Normally it is equal to 0 or to \( b \).

\( b \)

Array of size \( n \). Contains the right-hand side vector.

Output Parameters

\( RCI\_request \)

Gives information about result of the routine.

\( ipar \)

Array of size 128. Refer to the FGMRES Common Parameters.

\( dpar \)

Array of size 128. Refer to the FGMRES Common Parameters.

\( tmp \)


Return Values

\( RCI\_request = 0 \)

Indicates that the task completed normally.

\( RCI\_request = -1100 \)

Indicates that the task is interrupted and the errors occur.

\( RCI\_request = -1001 \)

Indicates that there are some warning messages.

\( RCI\_request = -1010 \)

Indicates that the routine changed some parameters to make them consistent or correct.

\( RCI\_request = -1011 \)

Indicates that there are some warning messages and that the routine changed some parameters.

\textit{dfgmres}

\textit{Makes the FGMRES iterations.}

Syntax

\[
\text{void dfgmres (const MKL_INT } *n, \text{ double } *x, \text{ double } *b, \text{ MKL_INT } *RCI\_request, \text{ MKL_INT } *ipar, \text{ double } *dpar, \text{ double } *tmp );}
\]

Include Files
• mkl.h
**Description**

The routine `dfgmres` performs the FGMRES iterations [Saad03], using the value that was in the array `x` before the first call as an initial approximation of the solution vector. To update the current approximation to the solution, the `dfgmres_get` routine must be called. The RCI FGMRES iterations can be continued after the call to the `dfgmres_get` routine only if the value of the parameter `ipar[12]` is not equal to 0 (default value). Note that the updated solution overwrites the right-hand side in the vector `b` if the parameter `ipar[12]` is positive, and the restarted version of the FGMRES method can not be run. If you want to keep the right-hand side, you must be save it in a different memory location before the first call to the `dfgmres_get` routine with a positive `ipar[12].`

The parameter `RCI_request` gives information about the task completion and requests results of certain operations that the solver requires.

The lengths of all the vectors must be defined in a previous call to the `dfgmres_init` routine.

**Input Parameters**

- `n`  
  Sets the size of the problem.

- `x`  
  Array of size `n`. Contains the initial approximation to the solution vector.

- `b`  
  Array of size `n`. Contains the right-hand side vector.

- `tmp`  
  Array of size `[12]`. Refer to the FGMRES Common Parameters.

**Output Parameters**

- `RCI_request`  
  Informs about result of work of the routine.

- `ipar`  
  Array of size `128`. Refer to the FGMRES Common Parameters.

- `dpar`  
  Array of size `128`. Refer to the FGMRES Common Parameters.

- `tmp`  
  Array of size `((2*ipar(15)+1)*n+ipar(15)*ipar(15)+9)/2 + 1)`. Refer to the FGMRES Common Parameters.

**Return Values**

- `RCI_request=0`  
  Indicates that the task completed normally and the solution is found and stored in the vector `x`. This occurs only if the stopping tests are fully automatic. For the user defined stopping tests, see the description of the `RCI_request= 2 or 4`.

- `RCI_request=-1`  
  Indicates that the routine was interrupted because the maximum number of iterations was reached, but the relative stopping criterion was not met. This situation occurs only if you request both tests.

- `RCI_request= -10`  
  Indicates that the routine was interrupted because of an attempt to divide by zero. Usually this happens if the matrix is degenerate or almost degenerate. However, it may happen if the parameter `dpar` is altered, or if the method is not stopped when the solution is found.
$RCI_{-request} = -11$

Indicates that the routine was interrupted because it entered an infinite cycle. Usually this happens because the values $ipar[7], ipar[8], ipar[9]$ were altered outside of the routine, or the $dfgmres$ routine was not called.

$RCI_{-request} = -12$

Indicates that the routine was interrupted because errors were found in the method parameters. Usually this happens if the parameters $ipar$ and $dpar$ were altered by mistake outside the routine.

$RCI_{-request} = 1$


$RCI_{-request} = 2$

Indicates that you must perform the stopping tests. If they fail, return control to the $dfgmres$ routine. Otherwise, the FGMRES solution is found, and you can run the $fgmres$ routine to update the computed solution in the vector $x$.

$RCI_{-request} = 3$


$RCI_{-request} = 4$

Indicates that you must check the norm of the currently generated vector. If it is not zero within the computational/rounding errors, return control to the $dfgmres$ routine. Otherwise, the FGMRES solution is found, and you can run the $dfgmres$ routine to update the computed solution in the vector $x$.

dfgmres_get

Retrieves the number of the current iteration and updates the solution.

**Syntax**

```c
void dfgmres_get (const MKL_INT *n, double *x, double *b, MKL_INT *RCI_request, const MKL_INT *ipar, const double *dpar, double *tmp, MKL_INT *itercount );
```

**Include Files**

- mkl.h

**Description**

The routine $dfgmres_get$ retrieves the current iteration number of the solution process and updates the solution according to the computations performed by the $dfgmres$ routine. To retrieve the current iteration number only, set the parameter $ipar[12] = -1$ beforehand. Normally, you should do this before proceeding further with the computations. If the intermediate solution is needed, the method parameters must be set properly. For details see FGMRES Common Parameters and the Iterative Sparse Solver code examples in the Intel® MKL installation directory:

- examples/solverc/source
Input Parameters

- \( n \) Sets the size of the problem.
- \( ipar \) Array of size 128. Refer to the FGMRES Common Parameters.
- \( dpar \) Array of size 128. Refer to the FGMRES Common Parameters.

Output Parameters

- \( x \) Array of size \( n \). If \( ipar[12] = 0 \), it contains the updated approximation to the solution according to the computations done in \texttt{dfgmres} routine. Otherwise, it is not changed.
- \( b \) Array of size \( n \). If \( ipar[13] > 0 \), it contains the updated approximation to the solution according to the computations done in \texttt{dfgmres} routine. Otherwise, it is not changed.
- \( RCI\_request \) Gives information about result of the routine.
- \( icount \) Contains the value of the current iteration number.

Return Values

- \( RCI\_request = 0 \) Indicates that the task completed normally.
- \( RCI\_request = -12 \) Indicates that the routine was interrupted because errors were found in the routine parameters. Usually this happens if the parameters \( ipar \) and \( dpar \) were altered by mistake outside of the routine.
- \( RCI\_request = -10000 \) Indicates that the routine failed to complete the task.

RCI ISS Implementation Details

Several aspects of the Intel® MKL RCI ISS interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, include one of the Intel® MKL RCI ISS language-specific header files.

NOTE

Intel® MKL does not support the RCI ISS interface unless you include the language-specific header file.

Preconditioners based on Incomplete LU Factorization Technique

Preconditioners, or accelerators are used to accelerate an iterative solution process. In some cases, their use can reduce the number of iterations dramatically and thus lead to better solver performance. Although the terms \emph{preconditioner} and \emph{accelerator} are synonyms, hereafter only \emph{preconditioner} is used.
Intel® MKL provides two preconditioners, ILU0 and ILUT, for sparse matrices presented in the format accepted in the Intel® MKL direct sparse solvers (three-array variation of the CSR storage format described in Sparse Matrix Storage Format). The algorithms used are described in [Saad03].

The ILU0 preconditioner is based on a well-known factorization of the original matrix into a product of two triangular matrices: lower and upper triangular matrices. Usually, such decomposition leads to some fill-in in the resulting matrix structure in comparison with the original matrix. The distinctive feature of the ILU0 preconditioner is that it preserves the structure of the original matrix in the result.

Unlike the ILU0 preconditioner, the ILUT preconditioner preserves some resulting fill-in in the preconditioner matrix structure. The distinctive feature of the ILUT algorithm is that it calculates each element of the preconditioner and saves each one if it satisfies two conditions simultaneously: its value is greater than the product of the given tolerance and matrix row norm, and its value is in the given bandwidth of the resulting preconditioner matrix.

Both ILU0 and ILUT preconditioners can apply to any non-degenerate matrix. They can be used alone or together with the Intel® MKL RCI FGMRES solver (see Sparse Solver Routines). Avoid using these preconditioners with MKL RCI CG solver because in general, they produce a non-symmetric resulting matrix even if the original matrix is symmetric. Usually, an inverse of the preconditioner is required in this case. To do this the Intel® MKL triangular solver routine mkl_dcsrtrsv must be applied twice: for the lower triangular part of the preconditioner, and then for its upper triangular part.

NOTE
Although ILU0 and ILUT preconditioners apply to any non-degenerate matrix, in some cases the algorithm may fail to ensure successful termination and the required result. Whether or not the preconditioner produces an acceptable result can only be determined in practice.

A preconditioner may increase the number of iterations for an arbitrary case of the system and the initial solution, and even ruin the convergence. It is your responsibility as a user to choose a suitable preconditioner.

General Scheme of Using ILUT and RCI FGMRES Routines

The general scheme for use is the same for both preconditioners. Some differences exist in the calling parameters of the preconditioners and in the subsequent call of two triangular solvers. You can see all these differences in the preconditioner code examples (dcsrilu*.*) in the examples folder of the Intel® MKL installation directory:

- examples/solverc/source

ILU0 and ILUT Preconditioners Interface Description

The concepts required to understand the use of the Intel® MKL preconditioner routines are discussed in the Appendix A Linear Solvers Basics.

User Data Arrays

The preconditioner routines take arrays of user data as input. To minimize storage requirements and improve overall run-time efficiency, the Intel® MKL preconditioner routines do not make copies of the user input arrays.

Common Parameters

Some parameters of the preconditioners are common with the FGMRES Common Parameters. The routine dfgmres_init specifies their default and initial values. However, some parameters can be redefined with other values. These parameters are listed below.

For the ILU0 preconditioner:
ipar[1] - specifies the destination of error messages generated by the ILU0 routine. The default value 6 means that all error messages are displayed on the screen. Otherwise routine creates a log file called MKL_PREC_log.txt and writes error messages to it. Note if the parameter ipar[5] is set to 0, then error messages are not generated at all.

ipar[5] - specifies whether error messages are generated. If its value is not equal to 0, the ILU0 routine returns error messages as specified by the parameter ipar[1]. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter ierr. The default value is 1.

For the ILUT preconditioner:

ipar[1] - specifies the destination of error messages generated by the ILUT routine. The default value 6 means that all messages are displayed on the screen. Otherwise routine creates a log file called MKL_PREC_log.txt and writes error messages to it. Note if the parameter ipar[5] is set to 0, then error messages are not generated at all.

ipar[5] - specifies whether error messages are generated. If its value is not equal to 0, the ILUT routine returns error messages as specified by the parameter ipar[1]. Otherwise, the routine does not generate error messages at all, but returns a negative value for the parameter ierr. The default value is 1.

ipar[6] - if its value is greater than 0, the ILUT routine generates warning messages as specified by the parameter ipar[1] and continues calculations. If its value is equal to 0, the routine returns a positive value of the parameter ierr. If its value is less than 0, the routine generates a warning message as specified by the parameter ipar[1] and returns a positive value of the parameter ierr. The default value is 1.

dcsrilu0

ILU0 preconditioner based on incomplete LU factorization of a sparse matrix.

Syntax

void dcsrilu0 (const MKL_INT *n, const double *a, const MKL_INT *ia, const MKL_INT *ja, double *bilu0, const MKL_INT *ipar, const double *dpar, MKL_INT *ierr);

Include Files

• mkl.h

Description

The routine dcsrilu0 computes a preconditioner $B$ [Saad03] of a given sparse matrix $A$ stored in the format accepted in the direct sparse solvers:

$A = B = L^*U$, where $L$ is a lower triangular matrix with a unit diagonal, $U$ is an upper triangular matrix with a non-unit diagonal, and the portrait of the original matrix $A$ is used to store the incomplete factors $L$ and $U$.

Caution

This routine supports only one-based indexing of the array parameters.

Input Parameters

$n$

Size (number of rows or columns) of the original square $n$-by-$n$ matrix $A$.

$a$

Array containing the set of elements of the matrix $A$. Its length is equal to the number of non-zero elements in the matrix $A$. Refer to the values array description in the Sparse Matrix Storage Format for more details.
**ia**
Array of size \((n+1)\) containing begin indices of rows of the matrix \(A\) such that \(ia[i]\) is the index in the array \(a\) of the first non-zero element from the row \(i\). The value of the last element \(ia[n]\) is equal to the number of non-zero elements in the matrix \(A\), plus one. Refer to the `rowIndex` array description in the Sparse Matrix Storage Format for more details.

**ja**
Array containing the column indices for each non-zero element of the matrix \(A\). It is important that the indices are in increasing order per row. The matrix size is equal to the size of the array \(a\). Refer to the `columns` array description in the Sparse Matrix Storage Format for more details.

**Caution**
If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

**ipar**
Array of size 128. This parameter specifies the integer set of data for both the ILU0 and RCI FGMRES computations. Refer to the `ipar` array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries that are specific to ILU0 are listed below.

\[ ipar[30] \]
specifies how the routine operates when a zero diagonal element occurs during calculation. If this parameter is set to 0 (the default value set by the routine `dfgmres_init`), then the calculations are stopped and the routine returns a non-zero error value. Otherwise, the diagonal element is set to the value of `dpar[31]` and the calculations continue.

**NOTE**
You can declare the `ipar` array with a size of 32. However, for future compatibility you must declare the array `ipar` with length 128.

**dpar**
Array of size 128. This parameter specifies the double precision set of data for both the ILU0 and RCI FGMRES computations. Refer to the `dpar` array description in the FGMRES Common Parameters for more details on FGMRES parameter entries. The entries specific to ILU0 are listed below.

\[ dpar[30] \]
specifies a small value, which is compared with the computed diagonal elements. When `ipar[30]` is not 0, then diagonal elements less than `dpar[30]` are set to `dpar[31]`. The default value is \(1.0\times10^{-16}\).
NOTE
This parameter can be set to the negative value, because the calculation uses its absolute value.
If this parameter is set to 0, the comparison with the diagonal element is not performed.

$dpar[31]$ specifies the value that is assigned to the diagonal element if its value is less than $dpar[30]$ (see above). The default value is $1.0e-10$.

NOTE
You can declare the $dpar$ array with a size of 32. However, for future compatibility you must declare the array $dpar$ with length 128.

Output Parameters

$bilu0$
Array $B$ containing non-zero elements of the resulting preconditioning matrix $B$, stored in the format accepted in direct sparse solvers. Its size is equal to the number of non-zero elements in the matrix $A$. Refer to the values array description in the Sparse Matrix Storage Format section for more details.

$ierr$
Error flag, gives information about the routine completion.

NOTE
To present the resulting preconditioning matrix in the CSR3 format the arrays $ia$ (row indices) and $ja$ (column indices) of the input matrix must be used.

Return Values

$ierr=0$
Indicates that the task completed normally.

$ierr=-101$
Indicates that the routine was interrupted and that error occurred: at least one diagonal element is omitted from the matrix in CSR3 format (see Sparse Matrix Storage Format).

$ierr=-102$
Indicates that the routine was interrupted because the matrix contains a diagonal element with the value of zero.

$ierr=-103$
Indicates that the routine was interrupted because the matrix contains a diagonal element which is so small that it could cause an overflow, or that it would cause a bad approximation to ILU0.

$ierr=-104$
Indicates that the routine was interrupted because the memory is insufficient for the internal work array.
ierr=-105 Indicates that the routine was interrupted because the input matrix size \( n \) is less than or equal to 0.

ierr=-106 Indicates that the routine was interrupted because the column indices \( ja \) are not in the ascending order.

dcsrilut
ILUT preconditioner based on the incomplete LU factorization with a threshold of a sparse matrix.

Syntax
```c
void dcsrilut (const MKL_INT *n, const double *a, const MKL_INT *ia, const MKL_INT *ja, double *bilut, MKL_INT *ibilut, MKL_INT *jbilut, const double *tol, const MKL_INT *maxfil, const MKL_INT *ipar, const double *dpar, MKL_INT *ierr);
```

Include Files
- mkl.h

Description
The routine dcsrilut computes a preconditioner \( B \) [Saad03] of a given sparse matrix \( A \) stored in the format accepted in the direct sparse solvers:
\[
A - B = L * U,
\]

where \( L \) is a lower triangular matrix with unit diagonal and \( U \) is an upper triangular matrix with non-unit diagonal.

The following threshold criteria are used to generate the incomplete factors \( L \) and \( U \):

1) the resulting entry must be greater than the matrix current row norm multiplied by the parameter \( tol \), and

2) the number of the non-zero elements in each row of the resulting \( L \) and \( U \) factors must not be greater than the value of the parameter \( maxfil \).

Caution
This routine supports only one-based indexing of the array parameters.

Input Parameters

\( n \) Size (number of rows or columns) of the original square \( n \)-by-\( n \) matrix \( A \).

\( a \) Array containing all non-zero elements of the matrix \( A \). The length of the array is equal to their number. Refer to \textit{values} array description in the Sparse Matrix Storage Format section for more details.

\( ia \) Array of size \( (n+1) \) containing indices of non-zero elements in the array \( a \). \( ia[i] \) is the index of the first non-zero element from the row \( i \). The value of the last element \( ia[n] \) is equal to the number of non-zeros in the matrix \( A \), plus one. Refer to the \textit{rowIndex} array description in the Sparse Matrix Storage Format for more details.
ja

Array of size equal to the size of the array \( a \). This array contains the column numbers for each non-zero element of the matrix \( A \). It is important that the indices are in increasing order per row. Refer to the \textit{columns array} description in the \textit{Sparse Matrix Storage Format} for more details.

\textbf{Caution}

If column indices are not stored in ascending order for each row of matrix, the result of the routine might not be correct.

tol

Tolerance for threshold criterion for the resulting entries of the preconditioner.

maxfil

Maximum fill-in, which is half of the preconditioner bandwidth. The number of non-zero elements in the rows of the preconditioner cannot exceed \((2 \times \text{maxfil} + 1)\).

ipar

Array of size 128. This parameter is used to specify the integer set of data for both the ILUT and RCI FGMRES computations. Refer to the \textit{ipar array} description in the \textit{FGMRES Common Parameters} for more details on FGMRES parameter entries. The entries specific to ILUT are listed below.

\textit{ipar[30]}

specifies how the routine operates if the value of the computed diagonal element is less than the current matrix row norm multiplied by the value of the parameter \( \text{tol} \). If \( \text{ipar}[30] = 0 \), then the calculation is stopped and the routine returns non-zero error value. Otherwise, the value of the diagonal element is set to a value determined by \( \text{dpar}[30] \) (see its description below), and the calculations continue.

\textbf{NOTE}

There is no default value for \( \text{ipar}[30] \) even if the preconditioner is used within the RCI ISS context. Always set the value of this entry.

\textbf{NOTE}

You must declare the array \( \text{ipar} \) with length 128. While defining the array in the code as \( \text{ipar}[30] \) works, there is no guarantee of future compatibility with Intel® MKL.

dpar

Array of size 128. This parameter specifies the double precision set of data for both ILUT and RCI FGMRES computations. Refer to the \textit{dpar array} description in the \textit{FGMRES Common Parameters} for more details on FGMRES parameter entries. The entries that are specific to ILUT are listed below.
$dpar[30]$ is used to adjust the value of small diagonal elements. Diagonal elements with a value less than the current matrix row norm multiplied by $tol$ are replaced with the value of $dpar[30]$ multiplied by the matrix row norm.

**NOTE**
There is no default value for $dpar[30]$ entry even if the preconditioner is used within RCI ISS context. Always set the value of this entry.

**NOTE**
You must declare the array $dpar$ with length 128. While defining the array in the code as $ipar[30]$ works, there is no guarantee of future compatibility with Intel® MKL.

### Output Parameters

**bilut**

Array containing non-zero elements of the resulting preconditioning matrix $B$, stored in the format accepted in the direct sparse solvers. Refer to the values array description in the Sparse Matrix Storage Format for more details. The size of the array is equal to $(2 \times \text{maxfil} + 1) \times n - \text{maxfil} \times (\text{maxfil} + 1) + 1$.

**NOTE**
Provide enough memory for this array before calling the routine. Otherwise, the routine may fail to complete successfully with a correct result.

**ibilut**

Array of size $(n+1)$ containing indices of non-zero elements in the array $bilut$. $ibilut[i]$ is the index of the first non-zero element from the row $i$. The value of the last element $ibilut[n]$ is equal to the number of non-zeros in the matrix $B$, plus one. Refer to the rowIndex array description in the Sparse Matrix Storage Format for more details.

**jbilut**

Array, its size is equal to the size of the array $bilut$. This array contains the column numbers for each non-zero element of the matrix $B$. Refer to the columns array description in the Sparse Matrix Storage Format for more details.

**ierr**

Error flag, gives information about the routine completion.

### Return Values

$ierr=0$ Indicates that the task completed normally.
$ierr=-101$
Indicates that the routine was interrupted because of an error: the number of elements in some matrix row specified in the sparse format is equal to or less than 0.

$ierr=-102$
Indicates that the routine was interrupted because the value of the computed diagonal element is less than the product of the given tolerance and the current matrix row norm, and it cannot be replaced as $ipar[30]=0$.

$ierr=-103$
Indicates that the routine was interrupted because the element $ia[i]$ is less than or equal to the element $ia[i-1]$ (see Sparse Matrix Storage Format).

$ierr=-104$
Indicates that the routine was interrupted because the memory is insufficient for the internal work arrays.

$ierr=-105$
Indicates that the routine was interrupted because the input value of $maxfil$ is less than 0.

$ierr=-106$
Indicates that the routine was interrupted because the size $n$ of the input matrix is less than 0.

$ierr=-107$
Indicates that the routine was interrupted because an element of the array $ja$ is less than 1, or greater than $n$ (see Sparse Matrix Storage Format).

$ierr=101$
The value of $maxfil$ is greater than or equal to $n$. The calculation is performed with the value of $maxfil$ set to $(n-1)$.

$ierr=102$
The value of $tol$ is less than 0. The calculation is performed with the value of the parameter set to $(-tol)$

$ierr=103$
The absolute value of $tol$ is greater than value of $dpar[30]$; it can result in instability of the calculation.

$ierr=104$
The value of $dpar[30]$ is equal to 0. It can cause calculations to fail.

**Sparse Matrix Checker Routines**

Intel® MKL provides a sparse matrix checker so that you can find errors in the storage of sparse matrices before calling Intel® MKL PARDISO, DSS, or Sparse BLAS routines.

**sparse_matrix_checker**
Checks correctness of sparse matrix.

**Syntax**

```c
MKL_INT sparse_matrix_checker (sparse_struct* handle);
```

**Include Files**

- mkl.h
**Description**

The `sparse_matrix_checker` routine checks a user-defined array used to store a sparse matrix in order to detect issues which could cause problems in routines that require sparse input matrices, such as Intel® MKL PARDISO, DSS, or Sparse BLAS.

**Input Parameters**

`handle`  
Pointer to the data structure describing the sparse array to check.

**Return Values**

The routine returns a value `error`. Additionally, the `check_result` parameter returns information about where the error occurred, which can be used when `message_level` is `MKL_NO_PRINT`.

**Sparse Matrix Checker Error Values**

<table>
<thead>
<tr>
<th><code>error value</code></th>
<th>Meaning</th>
<th>Location</th>
</tr>
</thead>
</table>
| MKL_SPARSE_CHECKER_SUCCESS | The input array successfully passed all checks. | C: `ia[i]` and `ia[i + 1]` are incompatible.  
`check_result[0] = i`  
`check_result[1] = ia[i]`  
`check_result[2] = ia[i + 1]` |
| MKL_SPARSE_CHECKER_NON_MONOTONIC | The input array is not 0 or 1 based (, `ia[0]` is not 0 or 1) or elements of `ia` are not in non-decreasing order as required. | C: `ia[i]` and `ia[i + 1]` are incompatible.  
`check_result[0] = j`  
`check_result[1] = ia[i]`  
`check_result[2] = ia[i + 1]` |
| MKL_SPARSE_CHECKER_OUT_OF_RANGE | The value of the `ja` array is lower than the number of the first column or greater than the number of the last column. | C: `ia[i]` and `ia[i + 1]` are incompatible.  
`check_result[0] = j`  
`check_result[1] = ia[i]`  
`check_result[2] = ia[i + 1]` |
| MKL_SPARSE_CHECKER_NON_TRIANGULAR | The `matrix_structure` parameter is MKL_UPPER_TRIANGULAR and both `ia` and `ja` are not upper triangular, or the `matrix_structure` parameter is MKL_LOWER_TRIANGULAR and both `ia` and `ja` are not lower triangular | C: `ia[i]` and `ja[j]` are incompatible.  
`check_result[0] = i`  
`check_result[1] = ia[i]`  
`check_result[2] = ja[j]` |
| MKL_SPARSE_CHECKER_NON_ORDERED | The elements of the `ja` array are not in non-decreasing order in each row as required. | C: `ia[i]` and `ia[i + 1]` are incompatible.  
`check_result[0] = j`  
`check_result[1] = ja[j]`  
`check_result[2] = ja[j + 1]` |
See Also

sparse_matrix_checker_init  Initializes handle for sparse matrix checker.

Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface
Sparse BLAS Level 2 and Level 3 Routines
Sparse Matrix Storage Formats

sparse_matrix_checker_init
Initializes handle for sparse matrix checker.

Syntax

```c
void sparse_matrix_checker_init (sparse_struct* handle);
```

Include Files

- mkl.h

Description

The `sparse_matrix_checker_init` routine initializes the handle for the `sparse_matrix_checker` routine. The `handle` variable contains this data:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Possible Values</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>MKL_INT</td>
<td></td>
<td>Order of the matrix stored in sparse array.</td>
</tr>
<tr>
<td>csr_ia</td>
<td>MKL_INT*</td>
<td>Pointer to <code>ia</code> array for <code>matrix_format</code> = MKL_CSR</td>
<td></td>
</tr>
<tr>
<td>csr_ja</td>
<td>MKL_INT*</td>
<td>Pointer to <code>ja</code> array for <code>matrix_format</code> = MKL_CSR</td>
<td></td>
</tr>
<tr>
<td>check_result[3]</td>
<td>MKL_INT</td>
<td>See Sparse Matrix Checker Error Values for a description of the values returned in <code>check_result</code>.</td>
<td>Indicates location of problem in array when <code>message_level</code> = MKL_NO_PRINT.</td>
</tr>
<tr>
<td>indexing</td>
<td>sparse_matrix_index</td>
<td>MKL_ZERO_BASED, MKL_ONE_BASED</td>
<td>Indexing style used in array.</td>
</tr>
<tr>
<td>matrix_structure</td>
<td>sparse_matrix_structures</td>
<td>MKL_GENERAL_STRUCTURE, MKL_UPPER_TRIANGULAR, MKL_LOWER_TRIANGULAR, MKL_STRUCTURAL_SYMMETRIC</td>
<td>Type of sparse matrix stored in array.</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Possible Values</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td>-----------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>matrix_format</td>
<td>sparse_matrix_formats</td>
<td>MKL_CSR</td>
<td>Format of array used for sparse matrix storage.</td>
</tr>
<tr>
<td>message_level</td>
<td>sparse_matrix_message_levels</td>
<td>MKL_NO_PRINT, MKL_PRINT</td>
<td>Determines whether or not feedback is provided on the screen.</td>
</tr>
<tr>
<td>print_style</td>
<td>sparse_matrix_print_styles</td>
<td>MKL_C_STYLE, MKL_FORTRAN_STYLE</td>
<td>Determines style of messages when message_level = MKL_PRINT.</td>
</tr>
</tbody>
</table>

**Input Parameters**

handle

Pointer to the data structure describing the sparse array to check.

**Output Parameters**

handle

Pointer to the initialized data structure.

**See Also**

sparse_matrix_checker Checks correctness of sparse matrix.
Intel® MKL PARDISO - Parallel Direct Sparse Solver Interface
Sparse BLAS Level 2 and Level 3 Routines
Sparse Matrix Storage Formats
Intel® MKL only supports the shared memory programming (SMP) version of the eigenvalue solver.

- The FEAST Algorithm gives a brief description of the algorithm underlying the Extended Eigensolver.
- Extended Eigensolver Functionality describes the problems that can and cannot be solved with the Extended Eigensolver and how to get the best results from the routines.
- Extended Eigensolver Interfaces gives a reference for calling Extended Eigensolver routines.

### The FEAST Algorithm

The Extended Eigensolver functionality is a set of high-performance numerical routines for solving symmetric standard eigenvalue problems, $Ax=\lambda x$, or generalized symmetric-definite eigenvalue problems, $Ax=\lambda Bx$. It yields all the eigenvalues ($\lambda$) and eigenvectors ($x$) within a given search interval $[\lambda_{\text{min}}, \lambda_{\text{max}}]$. It is based on the FEAST algorithm, an innovative fast and stable numerical algorithm presented in [Polizzi09], which fundamentally differs from the traditional Krylov subspace iteration based techniques (Arnoldi and Lanczos algorithms [Bai00]) or other Davidson-Jacobi techniques [Sleijpen96]. The FEAST algorithm is inspired by the density-matrix representation and contour integration techniques in quantum mechanics.

The FEAST numerical algorithm obtains eigenpair solutions using a numerically efficient contour integration technique. The main computational tasks in the FEAST algorithm consist of solving a few independent linear systems along the contour and solving a reduced eigenvalue problem. Consider a circle centered in the middle of the search interval $[\lambda_{\text{min}}, \lambda_{\text{max}}]$. The numerical integration over the circle in the current version of FEAST is performed using $N_e$-point Gauss-Legendre quadrature with $x_e$ the $e$-th Gauss node associated with the weight $\omega_e$. For example, for the case $N_e = 8$:

- $(x_1, \omega_1) = (0.183434642495649, 0.362683783378361),
- (x_2, \omega_2) = (-0.183434642495649, 0.362683783378361),
- (x_3, \omega_3) = (0.525532409916328, 0.31370645877887),
- (x_4, \omega_4) = (-0.525532409916328, 0.31370645877887),
- (x_5, \omega_5) = (0.796666477413626, 0.222381034453374),
- (x_6, \omega_6) = (-0.796666477413626, 0.222381034453374),
- (x_7, \omega_7) = (0.960289856497536, 0.101228536290376), and
- (x_8, \omega_8) = (-0.960289856497536, 0.101228536290376).

The figure FEAST Pseudocode shows the basic pseudocode for the FEAST algorithm for the case of real symmetric (left pane) and complex Hermitian (right pane) generalized eigenvalue problems, using $N$ for the size of the system and $M$ for the number of eigenvalues in the search interval (see [Polizzi09]).

**NOTE**

The pseudocode presents a simplified version of the actual algorithm. Refer to http://arxiv.org/abs/1302.0432 for an in-depth presentation and mathematical proof of convergence of FEAST.

### FEAST Pseudocode

\[
\begin{align*}
A & : \text{real symmetric} & A & : \text{complex Hermitian} \\
B & : \text{symmetric positive definite (SPD)} & B & : \text{Hermitian positive definite (HPD)} \\
\Re\{x\} & : \text{real part of } x
\end{align*}
\]
Extended Eigensolver Functionality

The eigenvalue problems covered are as follows:

- **standard**, \( Ax = \lambda x \)
  - A complex Hermitian
  - A real symmetric

- **generalized**, \( Ax = \lambda Bx \)
  - A complex Hermitian, \( B \) Hermitian positive definite (hpd)
  - A real symmetric and \( B \) real symmetric positive definite (spd)

The Extended Eigensolver functionality offers:

- **Real/Complex and Single/Double precisions**: double precision is recommended to provide better accuracy of eigenpairs.
- **Reverse communication interfaces (RCI)** provide maximum flexibility for specific applications. RCI are independent of matrix format and inner system solvers, so you must provide your own linear system solvers (direct or iterative) and matrix-matrix multiply routines.
- **Predefined driver interfaces** for dense, LAPACK banded, and sparse (CSR) formats are less flexible but are optimized and easy to use:
  - The Extended Eigensolver interfaces for dense matrices are likely to be slower than the comparable LAPACK routines because the FEAST algorithm has a higher computational cost.
  - The Extended Eigensolver interfaces for banded matrices support banded LAPACK-type storage.
  - The Extended Eigensolver sparse interfaces support compressed sparse row format and use the Intel® MKL PARDISO solver.
Parallelism in Extended Eigensolver Routines

How you achieve parallelism in Extended Eigensolver routines depends on which interface you use. Parallelism (via shared memory programming) is not explicitly implemented in Extended Eigensolver routines within one node: the inner linear systems are currently solved one after another.

- Using the Extended Eigensolver RCI interfaces, you can achieve parallelism by providing a threaded inner system solver and a matrix-matrix multiplication routine. When using the RCI interfaces, you are responsible for activating the threaded capabilities of your BLAS and LAPACK libraries most likely using the shell variable `OMP_NUM_THREADS`.
- Using the predefined Extended Eigensolver interfaces, parallelism can be implicitly obtained within the shared memory version of BLAS, LAPACK or Intel® MKL PARDISO. The shell variable `MKL_NUM_THREADS` can be used for automatically setting the number of OpenMP threads (cores) for BLAS, LAPACK, and Intel® MKL PARDISO.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Achieving Performance With Extended Eigensolver Routines

In order to use the Extended Eigensolver Routines, you need to provide

- the search interval and the size of the subspace $M_0$ (overestimation of the number of eigenvalues $M$ within a given search interval);
- the system matrix in dense, banded, or sparse CSR format if the Extended Eigensolver predefined interfaces are used, or a high-performance complex direct or iterative system solver and matrix-vector multiplication routine if RCI interfaces are used.

In return, you can expect

- fast convergence with very high accuracy when seeking up to 1000 eigenpairs (in two to four iterations using $M_0 = 1.5M$, and $N_e = 8$ or at most using $N_e = 16$ contour points);
- an extremely robust approach.

The performance of the basic FEAST algorithm depends on a trade-off between the choices of the number of Gauss quadrature points $N_e$, the size of the subspace $M_0$, and the number of outer refinement loops to reach the desired accuracy. In practice you should use $M_0 > 1.5$ $M$, $N_e = 8$, and at most two refinement loops.

For better performance:
• $M_0$ should be much smaller than the size of the eigenvalue problem, so that the arithmetic complexity depends mainly on the inner system solver ($O(NM)$ for narrow-banded or sparse systems).
• Parallel scalability performance depends on the shared memory capabilities of the of the inner system solver.
• For very large sparse and challenging systems, application users should make use of the Extended Eigensolver RCI interfaces with customized highly-efficient iterative systems solvers and preconditioners.
• For the Extended Eigensolver interfaces for banded matrices, the parallel performance scalability is limited.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

## Extended Eigensolver Interfaces for Eigenvalues within Interval

### Extended Eigensolver Naming Conventions

There are two different types of interfaces available in the Extended Eigensolver routines:

1. The reverse communication interfaces (RCI):

   ```
   ?feast_<matrix type>_rci
   ```

   These interfaces are matrix free format (the interfaces are independent of the matrix data formats). You must provide matrix-vector multiply and direct/iterative linear system solvers for your own explicit or implicit data format.

2. The predefined interfaces:

   ```
   ?feast_<matrix type><type of eigenvalue problem>
   ```

   are predefined drivers for ?feast reverse communication interface that act on commonly used matrix data storage (dense, banded and compressed sparse row representation), using internal matrix-vector routines and selected inner linear system solvers.

For these interfaces:

- ? indicates the data type of matrix $A$ (and matrix $B$ if any) defined as follows:
  - s float
  - d double
  - c MKL_Complex8
  - z MKL_Complex16

- <matrix type> defined as follows:
### Value of `<matrix type>`

<table>
<thead>
<tr>
<th><code>&lt;matrix type&gt;</code></th>
<th>Matrix format</th>
<th>Inner linear system solver used by Extended Eigensolver</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sy</code></td>
<td>(symmetric real)</td>
<td>LAPACK dense solvers</td>
</tr>
<tr>
<td><code>he</code></td>
<td>(Hermitian complex)</td>
<td>Banded-LAPACK</td>
</tr>
<tr>
<td><code>sb</code></td>
<td>(symmetric banded real)</td>
<td>Internal banded solver</td>
</tr>
<tr>
<td><code>hb</code></td>
<td>(Hermitian banded complex)</td>
<td>PARDISO solver</td>
</tr>
<tr>
<td><code>scsr</code></td>
<td>(symmetric real)</td>
<td>Reverse communications interfaces</td>
</tr>
<tr>
<td><code>hcsr</code></td>
<td>(Hermitian complex)</td>
<td>User defined</td>
</tr>
</tbody>
</table>

- `<type of eigenvalue problem>` is:
  - `gv`: generalized eigenvalue problem
  - `ev`: standard eigenvalue problem

For example, `sfeast_scsrev` is a single-precision routine with a symmetric real matrix stored in sparse compressed-row format for a standard eigenvalue problem, and `zfeast_hrci` is a complex double-precision routine with a Hermitian matrix using the reverse communication interface.

Note that:
- `?` can be `s` or `d` if a matrix is real symmetric: `<matrix type>` is `sy`, `sb`, or `scsr`.
- `?` can be `c` or `z` if a matrix is complex Hermitian: `<matrix type>` is `he`, `hb`, or `hcsr`.
- `?` can be `c` or `z` if the Extended Eigensolver RCI interface is used for solving a complex Hermitian problem.
- `?` can be `s` or `d` if the Extended Eigensolver RCI interface is used for solving a real symmetric problem.

#### feastinit

*Initialize Extended Eigensolver input parameters with default values.*

**Syntax**

```c
feastinit (MKL_INT* fpm);
```

**Include Files**

- `mkl.h`

**Description**

This routine sets all Extended Eigensolver parameters to their default values.
Output Parameters

*fpm* Array, size 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

Extended Eigensolver Input Parameters

The input parameters for Extended Eigensolver routines are contained in an MKL_INT array named *fpm*. To call the Extended Eigensolver interfaces, this array should be initialized using the routine  *feastinit*.

<table>
<thead>
<tr>
<th>Param</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| *fpm[0]* | 0 | Specifies whether Extended Eigensolver routines print runtime status.  
  *fpm[0]=0* Extended Eigensolver routines do not generate runtime messages at all.  
  *fpm[0]=1* Extended Eigensolver routines print runtime status to the screen. |
| *fpm[1]* | 8 | The number of contour points $N_e = 8$ (see the description of FEAST algorithm). Must be one of \{3,4,5,6,8,10,12,16,20,24,32,40,48\}. |
| *fpm[2]* | 12 | Error trace double precision stopping criteria $\varepsilon$ ($\varepsilon = 10^{-fpm[2]}$). |
| *fpm[3]* | 20 | Maximum number of Extended Eigensolver refinement loops allowed. If no convergence is reached within *fpm[3]* refinement loops, Extended Eigensolver routines return *info*=2. |
| *fpm[4]* | 0 | User initial subspace. If *fpm[4]=0* then Extended Eigensolver routines generate initial subspace, if *fpm[4]=1* the user supplied initial subspace is used. |
| *fpm[5]* | 0 | Extended Eigensolver stopping test.  
  *fpm[5]=0* Extended Eigensolvers are stopped if this residual stopping test is satisfied:  
  - *generalized eigenvalue problem*:  
    \[
    \max_{j=1,\text{work}} \frac{\|A x_j - \lambda_j B x_j\|}{\max \left(\|P_{\text{min}}\|, \|P_{\text{max}}\|\right)} < \varepsilon
    \]
  - *standard eigenvalue problem*:  
    \[
    \max_{j=1,\text{work}} \frac{\|A x_j - \lambda_j x_j\|}{\max \left(\|P_{\text{min}}\|, \|P_{\text{max}}\|\right)} < \varepsilon
    \]
  where *mode* is the total number of eigenvalues found in the search interval and $\varepsilon = 10^{-fpm[6]}$ for real and complex or $\varepsilon = 10^{-fpm[2]}$ for double precision and double complex.  
  *fpm[5]=1* Extended Eigensolvers are stopped if this trace stopping test is satisfied:  
  \[
  \frac{|\text{trace}_j - \text{trace}_{j-1}|}{\max \left(\|P_{\text{min}}\|, \|P_{\text{max}}\|\right)} < \varepsilon
  \]
where \( \text{trace}_j \) denotes the sum of all eigenvalues found in the search interval \([emin, emax]\) at the \( j \)-th Extended Eigensolver iteration:

\[
\text{trace}_j = \sum_{i=1}^{M} \lambda_j^{(i)}
\]

| \( fpm[6] \) | 5 | Error trace single precision stopping criteria \((10^{-fpm[6]})\). |
| \( fpm[13] \) | 0 | Standard use for Extended Eigensolver routines. |
| \( fpm[13] \) | 0 | Non-standard use for Extended Eigensolver routines: return the computed eigenvectors subspace after one single contour integration. |
| \( fpm[26] \) | 0 | Specifies whether Extended Eigensolver routines check input matrices (applies to CSR format only). |
| \( fpm[26] \) | 0 | Extended Eigensolver routines do not check input matrices. |
| \( fpm[26] \) | 1 | Extended Eigensolver routines check input matrices. |
| \( fpm[27] \) | 0 | Check if matrix \( B \) is positive definite. Set \( fpm[27] = 1 \) to check if \( B \) is positive definite. |
| \( fpm[29] \) | - | Reserved for future use. |
| \( fpm[63] \) | 0 | Use the Intel® MKL PARDISO solver with the user-defined PARDISO iparm array settings. |

**NOTE**
This option can only be used by Extended Eigensolver Predefined Interfaces for Sparse Matrices.

\( fpm[63] = 0 \) Extended Eigensolver routines use the Intel® MKL PARDISO default iparm settings defined by calling the pardisoinit subroutine.

\( fpm[63] = 1 \) The values from \( fpm[64] \) to \( fpm[127] \) correspond to \( iparm[0] \) to \( iparm[63] \) respectively according to the formula \( fpm[64 + i] = iparm[i] \) for \( i = 0, 1, \ldots, 63 \).

**Extended Eigensolver Output Details**
Errors and warnings encountered during a run of the Extended Eigensolver routines are stored in an integer variable, \( info \). If the value of the output \( info \) parameter is not 0, either an error or warning was encountered. The possible return values for the \( info \) parameter along with the error code descriptions are given in the following table.

<p>| ( info ) | Classification | Description |
| 202 | Error | Problem with size of the system ( n ) ((n \leq 0)) |</p>
<table>
<thead>
<tr>
<th>info</th>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>201</td>
<td>Error</td>
<td>Problem with size of initial subspace ( m0 ) (( m0 \leq 0 ) or ( m0 &gt; n ))</td>
</tr>
<tr>
<td>200</td>
<td>Error</td>
<td>Problem with ( e_{\min}, e_{\max} ) (( e_{\min} \leq e_{\max} ))</td>
</tr>
<tr>
<td>(100+i)</td>
<td>Error</td>
<td>Problem with ( i )-th value of the input Extended Eigensolver parameter ( f_{pm}[i] ). Only the parameters in use are checked.</td>
</tr>
<tr>
<td>4</td>
<td>Warning</td>
<td>Successful return of only the computed subspace after call with ( f_{pm}[13] = 1 )</td>
</tr>
<tr>
<td>3</td>
<td>Warning</td>
<td>Size of the subspace ( m0 ) is too small (( m0 &lt; m ))</td>
</tr>
<tr>
<td>2</td>
<td>Warning</td>
<td>No Convergence (number of iteration loops &gt; ( f_{pm}[3] ))</td>
</tr>
<tr>
<td>1</td>
<td>Warning</td>
<td>No eigenvalue found in the search interval. See remark below for further details.</td>
</tr>
<tr>
<td>0</td>
<td>Successful exit</td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>Error</td>
<td>Internal error for allocation memory.</td>
</tr>
<tr>
<td>-2</td>
<td>Error</td>
<td>Internal error of the inner system solver. Possible reasons: not enough memory for inner linear system solver or inconsistent input.</td>
</tr>
<tr>
<td>-3</td>
<td>Error</td>
<td>Internal error of the reduced eigenvalue solver Possible cause: matrix ( B ) may not be positive definite. It can be checked by setting ( f_{pm}[27] = 1 ) before calling an Extended Eigensolver routine, or by using LAPACK routines.</td>
</tr>
<tr>
<td>-4</td>
<td>Error</td>
<td>Matrix ( B ) is not positive definite.</td>
</tr>
<tr>
<td>-(100+i)</td>
<td>Error</td>
<td>Problem with the ( i )-th argument of the Extended Eigensolver interface.</td>
</tr>
</tbody>
</table>

In some extreme cases the return value \( info=1 \) may indicate that the Extended Eigensolver routine has failed to find the eigenvalues in the search interval. This situation could arise if a very large search interval is used to locate a small and isolated cluster of eigenvalues (i.e. the dimension of the search interval is many orders of magnitude larger than the number of contour points. It is then either recommended to increase the number of contour points \( f_{pm}[1] \) or simply rescale more appropriately the search interval. Rescaling means the initial problem of finding all eigenvalues the search interval \([\lambda_{\min}, \lambda_{\max}]\) for the standard eigenvalue problem \( A \ x = \lambda \ x \) is replaced with the problem of finding all eigenvalues in the search interval \([\lambda_{\min}/t, \lambda_{\max}/t]\) for the standard eigenvalue problem \( (A/t) \ x = (\lambda/t) \ x \) where \( t \) is a scaling factor.

**Extended Eigensolver RCI Routines**

If you do not require specific linear system solvers or matrix storage schemes, you can skip this section and go directly to Extended Eigensolver Predefined Interfaces.

**Extended Eigensolver RCI Interface Description**

The Extended Eigensolver RCI interfaces can be used to solve standard or generalized eigenvalue problems, and are independent of the format of the matrices. As mentioned earlier, the Extended Eigensolver algorithm is based on the contour integration techniques of the matrix resolvent \( G(\sigma) = (\sigma B - A)^{-1} \) over a circle. For solving a generalized eigenvalue problem, Extended Eigensolver has to perform one or more of the following operations at each contour point denoted below by \( Z_e \):

- Factorize the matrix \( (Z_e * B - A) \)
- Solve the linear system \( (Z_e * B - A)X = Y \) or \( (Z_e * B - A)^H X = Y \) with multiple right hand sides, where \( H \) means transpose conjugate
• Matrix-matrix multiply $BX = Y$ or $AX = Y$

For solving a standard eigenvalue problem, replace the matrix $B$ with the identity matrix $I$.

The primary aim of RCI interfaces is to isolate these operations: the linear system solver, factorization of the matrix resolvent at each contour point, and matrix-matrix multiplication. This gives universality to RCI interfaces as they are independent of data structures and the specific implementation of the operations like matrix-vector multiplication or inner system solvers. However, this approach requires some additional effort when calling the interface. In particular, operations listed above are performed by routines that you supply on data structures that you find most appropriate for the problem at hand.

To initialize an Extended Eigensolver RCI routine, set the job indicator ($ijob$) parameter to the value -1. When the routine requires the results of an operation, it generates a special value of $ijob$ to indicate the operation that needs to be performed. The routine also returns $ze$, the coordinate along the complex contour, the values of array $work$ or $workc$, and the number of columns to be used. Your subroutine then must perform the operation at the given contour point $ze$, store the results in prescribed array, and return control to the Extended Eigensolver RCI routine.

The following pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a real symmetric problem:

```plaintext
ijob=-1; // initialization
do while (ijob!=0) {
    ?feast_srci(&ijob, &N, &ze, work, workc, Aq, Bq,
        fpm, &epsout, &loop, &Emin, &Emax, &M0, E, lambda, &q, res, &info);
    switch(ijob) {
        case 10: // Factorize the complex matrix (ZeB-A)
            break;
        case 11: // Solve the complex linear system (ZeB-A)x=workc
            // Put result in workc
            break;
        case 30: // Perform multiplication A by Qi..Qj columns of Q_{Nu00}
            // where i = fpm[23] and j = fpm[23]+fpm[24]-1
            // Qi..Qj located in q starting from q+N*(i-1)
            break;
        case 40: // Perform multiplication B by Qi..Qj columns of Q_{Nu00}
            // where i = fpm[23] and j = fpm[23]+fpm[24]-1
            // Qi..Qj located in q starting from q+N*(i-1)
            // Result is stored in work+N*(i-1)
            break;
    }
}
```

**NOTE**
The `?` option in `?feast` in the pseudocode given above should be replaced by either $s$ or $d$, depending on the matrix data type of the eigenvalue system.

The next pseudocode shows the general scheme for using the Extended Eigensolver RCI functionality for a complex Hermitian problem:

```plaintext
ijob=-1; // initialization
while (ijob!=0) {
    ?feast_hrci(&ijob, &N, &ze, work, workc, Aq, Bq,
        fpm, &epsout, &loop, &Emin, &Emax, &M0, E, lambda, &q, res, &info);
```
switch (ijob) {
    case 10: // Factorize the complex matrix (ZeB-A)
        break;

    case 11: // Solve the linear system (ZeB-A)y=workc
        // Put result in workc
        break;

    case 20: // Factorize (if needed by case 21) the complex matrix (ZeB-A)^H
        // ATTENTION: This option requires additional memory storage
        // (i.e. the resulting matrix from case 10 cannot be overwritten)
        break;

    case 21: // Solve the linear system (ZeB-A)^Hy=workc
        // Put result in workc
        // REMARK: case 20 becomes obsolete if this solve can be performed
        // using the factorization in case 10
        break;

    case 30: // Multiply A by Q_(i..j) columns of Q_{NxM0},
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Q_(i..j) located in q starting from q+N*(i-1)
        // Result is stored in work+N*(i-1)
        break;

    case 40: // Perform multiplication B by Q_(i..j) columns of Q_{NxM0}
        // where i = fpm[23] and j = fpm[23]+fpm[24]-1
        // Q_(i..j) located in q starting from q+N*(i-1)
        // Result is stored in work+N*(i-1)
        break;
    }
}
end do

NOTE
The ? option in ?feast in the pseudocode given above should be replaced by either c or z, depending on the matrix data type of the eigenvalue system.

If case 20 can be avoided, performance could be up to twice as fast, and Extended Eigensolver functionality would use half of the memory.

If an iterative solver is used along with a preconditioner, the factorization of the preconditioner could be performed with ijob = 10 (and ijob = 20 if applicable) for a given value of Ze, and the associated iterative solve would then be performed with ijob = 11 (and ijob = 21 if applicable).

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
**feast_srci/**feast_hrci

*Extended Eigensolver RCI interface.*

**Syntax**

```c
void sfeast_srci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, float* work, MKL_Complex8* workc, float* sq, MKL_INT* fpm, float* epsout, MKL_INT* loop, const float* emin, const float* emax, MKL_INT* m0, float* lambda, float* q, MKL_INT* m, float* res, MKL_INT* info);

void dfeast_srci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, double* work, MKL_Complex16* workc, double* sq, MKL_INT* fpm, double* epsout, MKL_INT* loop, const double* emin, const double* emax, MKL_INT* m0, double* lambda, double* q, MKL_INT* m, double* res, MKL_INT* info);

void cfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex8* ze, MKL_Complex8* work, MKL_Complex8* workc, MKL_Complex8* sq, MKL_INT* fpm, float* epsout, MKL_INT* loop, const float* emin, const float* emax, MKL_INT* m0, float* lambda, float* q, MKL_INT* m, float* res, MKL_INT* info);

void zfeast_hrci (MKL_INT* ijob, const MKL_INT* n, MKL_Complex16* ze, MKL_Complex16* work, MKL_Complex16* workc, MKL_Complex16* sq, MKL_INT* fpm, double* epsout, MKL_INT* loop, const double* emin, const double* emax, MKL_INT* m0, double* lambda, MKL_Complex16* q, MKL_INT* m, double* res, MKL_INT* info);
```

**Include Files**

- mkl.h

**Description**

Compute eigenvalues as described in *Extended Eigensolver RCI Interface Description*.  

**Input Parameters**

- **ijob**

  Job indicator variable. On entry, a call to `?feast_srci/?feast_hrci` with `ijob=-1` initializes the eigensolver.

- **n**

  Sets the size of the problem. \( n > 0 \).

- **work**

  Workspace array of size \( n \) by \( m0 \).

- **workc**

  Workspace array of size \( n \) by \( m0 \).

- **aq**, **sq**

  Workspace arrays of size \( m0 \) by \( m0 \).

- **fpm**

  Array, size of 128. This array is used to pass various parameters to Extended Eigensolver routines. See *Extended Eigensolver Input Parameters* for a complete description of the parameters and their default values.

- **emin**, **emax**

  The lower and upper bounds of the interval to be searched for eigenvalues; \( emin \leq emax \).

- **m0**

  On entry, specifies the initial guess for subspace size to be used, \( 0 < m0 \leq n \). Set \( m0 \geq m \) where \( m \) is the total number of eigenvalues located in the interval \([emin, emax]\). If the initial guess is wrong, Extended Eigensolver routines return `info`=3.
On entry, if \( fpm[4] = 1 \), the array \( q \) of size \( n \) by \( m \) contains a basis of guess subspace where \( n \) is the order of the input matrix.

Output Parameters

\( ijob \)

On exit, the parameter carries the status flag that indicates the condition of the return. The status information is divided into three categories:

1. A zero value indicates successful completion of the task.
2. A positive value indicates that the solver requires a matrix-vector multiplication or solving a specific system with a complex coefficient.
3. A negative value indicates successful initiation.

A non-zero value of \( ijob \) specifically means the following:

- \( ijob = 10 \) - factorize the complex matrix \( Z_e^* B - A \) at a given contour point \( Z_e \) and return the control to the \(?feast_srci/?feast_hrci\) routine where \( Z_e \) is a complex number meaning contour point and its value is defined internally in \(?feast_srci/?feast_hrci\).
- \( ijob = 11 \) - solve the complex linear system \((Z_e^* B - A)^* y = workc\), put the solution in \( workc \) and return the control to the \(?feast_srci/?feast_hrci\) routine.
- \( ijob = 20 \) - factorize the complex matrix \((Z_e^* B - A)^H\) at a given contour point \( Z_e \) and return the control to the \(?feast_srci/?feast_hrci\) routine where \( Z_e \) is a complex number meaning contour point and its value is defined internally in \(?feast_srci/?feast_hrci\).
- \( ijob = 21 \) - solve the complex linear system \((Z_e^* B - A)^H^* y = workc\), put the solution in \( workc \) and return the control to the \(?feast_srci/?feast_hrci\) routine. The case \( ijob=20 \) becomes obsolete if the solve can be performed using the factorization computed for \( ijob=10 \).
- \( ijob = 30 \) - multiply matrix \( A \) by \( Q_j..Q_i \), put the result in \( work + N^*(i - 1) \), and return the control to the \(?feast_srci/?feast_hrci\) routine.
  
  \( i \) is \( fpm[24] \), and \( j \) is \( fpm[23] + fpm[24] - 1 \).
- \( ijob = 40 \) - multiply matrix \( B \) by \( Q_j..Q_i \), put the result in \( work + N^*(i - 1) \) and return the control to the \(?feast_srci/?feast_hrci\) routine. If a standard eigenvalue problem is solved, just return \( work = q \).
  
  \( i \) is \( fpm[24] \), and \( j \) is \( fpm[23] + fpm[24] - 1 \).
- \( ijob = -2 \) - rerun the \(?feast_srci/?feast_hrci\) task with the same parameters.

\( ze \)

Defines the coordinate along the complex contour. All values of \( ze \) are generated by \(?feast_srci/?feast_hrci\) internally.

\( fpm \)

On output, contains coordinates of columns of work array needed for iterative refinement. (See Extended Eigensolver RCI Interface Description.)

\( epsout \)

On output, contains the relative error on the trace: \( |trace_i - trace_{i-1}| / \max(|emin|, |emax|) \)
On output, contains the number of refinement loop executed. Ignored on input.

Array of length \( m_0 \). On output, the first \( m \) entries of \( \lambda \) are eigenvalues found in the interval.

On output, \( q \) contains all eigenvectors corresponding to \( \lambda \).

The total number of eigenvalues found in the interval \([emin, emax]\): 0 \( \leq m \leq m_0 \).

Array of length \( m_0 \). On exit, the first \( m \) components contain the relative residual vector:

- generalized eigenvalue problem:
  \[
  \frac{\|A\tilde{x}_i - \lambda_i\tilde{x}_i\|}{\max(\|E_{max}\|, \|E_{min}\|)}\|\tilde{x}_i\|
  \]

- standard eigenvalue problem:
  \[
  \frac{\|A\tilde{x}_i - \tilde{x}_i\|}{\max(\|E_{max}\|, \|E_{min}\|)}\|\tilde{x}_i\|
  \]

for \( i = 0, 1, \ldots, m - 1 \), and where \( m \) is the total number of eigenvalues found in the search interval.

If \( info = 0 \), the execution is successful. If \( info \neq 0 \), see Output Eigensolver info Details.

## Extended Eigensolver Predefined Interfaces

The predefined interfaces include routines for standard and generalized eigenvalue problems, and for dense, banded, and sparse matrices.

<table>
<thead>
<tr>
<th>Matrix Type</th>
<th>Standard Eigenvalue Problem</th>
<th>Generalized Eigenvalue Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dense</td>
<td>?feast_syev</td>
<td>?feast_sygv</td>
</tr>
<tr>
<td></td>
<td>?feast_heev</td>
<td>?feast_hegv</td>
</tr>
<tr>
<td>Banded</td>
<td>?feast_sbev</td>
<td>?feast_sbgv</td>
</tr>
<tr>
<td></td>
<td>?feast_hbev</td>
<td>?feast_hbgv</td>
</tr>
<tr>
<td>Sparse</td>
<td>?feast_scsrev</td>
<td>?feast_scsrgv</td>
</tr>
<tr>
<td></td>
<td>?feast_hcsrev</td>
<td>?feast_hcsrgv</td>
</tr>
</tbody>
</table>

## Matrix Storage

The symmetric and Hermitian matrices used in Extended Eigensolvers predefined interfaces can be stored in full, band, and sparse formats.

- In the full storage format (described in Full Storage in additional detail) you store all elements, all of the elements in the upper triangle of the matrix, or all of the elements in the lower triangle of the matrix.
- In the band storage format (described in Band storage in additional detail), you store only the elements along a diagonal band of the matrix.
- In the sparse format (described in Storage Arrays for a Matrix in CSR Format (3-Array Variation)), you store only the non-zero elements of the matrix.
In generalized eigenvalue systems you must use the same family of storage format for both matrices \( A \) and \( B \). The bandwidth can be different for the banded format (\( k_{lb} \) can be different from \( k_{la} \)), and the position of the non-zero elements can also be different for the sparse format (CSR coordinates \( ib \) and \( jb \) can be different from \( ia \) and \( ja \)).

\texttt{?feast\_syev/?feast\_heev}  
Extended Eigensolver interface for standard eigenvalue problem with dense matrices.

**Syntax**

```c
void sfeast\_syev (const char * uplo, const MKL\_INT * n, const float * a, const MKL\_INT * lda, MKL\_INT * fpm, float * epsout, MKL\_INT * loop, const float * emin, const float * emax, MKL\_INT * m0, float * e, float * x, MKL\_INT * m, float * res, MKL\_INT * info);

void dfeast\_syev (const char * uplo, const MKL\_INT * n, const double * a, const MKL\_INT * lda, MKL\_INT * fpm, double * epsout, MKL\_INT * loop, const double * emin, const double * emax, MKL\_INT * m0, double * e, double * x, MKL\_INT * m, double * res, MKL\_INT * info);

void cfeast\_heev (const char * uplo, const MKL\_INT * n, const MKL\_Complex8 * a, const MKL\_INT * lda, MKL\_INT * fpm, float * epsout, MKL\_INT * loop, const float * emin, const float * emax, MKL\_INT * m0, float * e, MKL\_Complex8 * x, MKL\_INT * m, float * res, MKL\_INT * info);

void zfeast\_heev (const char * uplo, const MKL\_INT * n, const MKL\_Complex16 * a, const MKL\_INT * lda, MKL\_INT * fpm, double * epsout, MKL\_INT * loop, const double * emin, const double * emax, MKL\_INT * m0, double * e, MKL\_Complex16 * x, MKL\_INT * m, double * res, MKL\_INT * info);
```

**Include Files**

- mkl.h

**Description**

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, \( Ax = \lambda x \), within a given search interval.

**Input Parameters**

**uplo**

Must be ‘U’ or ‘L’ or ‘F’.

If \( uplo = 'U' \), \( a \) stores the upper triangular parts of \( A \).

If \( uplo = 'L' \), \( a \) stores the lower triangular parts of \( A \).

If \( uplo = 'F' \), \( a \) stores the full matrix \( A \).

**n**

Sets the size of the problem. \( n > 0 \).

**a**

Array of dimension \( lda \) by \( n \), contains either full matrix \( A \) or upper or lower triangular part of the matrix \( A \), as specified by \( uplo \).

**lda**

The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

**fpm**

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.
emin,emax

The lower and upper bounds of the interval to be searched for eigenvalues; 
emin ≤ emax.

m0

On entry, specifies the initial guess for subspace dimension to be used, 0 < m0< n. Set m0 ≥ m where m is the total number of eigenvalues located in the interval [emin,emax]. If the initial guess is wrong, Extended Eigensolver routines return info=3.

x

On entry, if fpm[4]=1, the array x of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

 epsout

On output, contains the relative error on the trace: |tracei - tracei-1| /max(|emin|, |emax|)

 loop

On output, contains the number of refinement loop executed. Ignored on input.

e

Array of length m0. On output, the first m entries of e are eigenvalues found in the interval.

x

On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e, with the i-th column of x holding the eigenvector associated with ei.

m

The total number of eigenvalues found in the interval [emin,emax]: 0 ≤ m ≤ m0.

res

Array of length m0. On exit, the first m components contain the relative residual vector:

\[ \frac{\|Ax_i - \lambda_i x_i\|}{\max(\|F_{\text{rel}}\|, \|F_{\text{rel}}\|)\|x_i\|} \]

for i=1, 2, ..., m, and where m is the total number of eigenvalues found in the search interval.

info

If info=0, the execution is successful. If info ≠ 0, see Output Eigensolver info Details.

?feast_sygv/?feast_hegv

Extended Eigensolver interface for generalized eigenvalue problem with dense matrices.

Syntax

void sfeast_sygv (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT * lda, const float * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sygv (const char * uplo, const MKL_INT * n, const double * a, const MKL_INT * lda, const double * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT * info);
void cfeast_hegv (const char * uplo, const MKL_INT * n, const MKL_Complex8 * a, const MKL_INT * lda, const MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);
void zfeast_hegv (const char * uplo, const MKL_INT * n, const MKL_Complex16 * a, const MKL_INT * lda, const MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);

Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, \( Ax = \lambda Bx \), within a given search interval.

Input Parameters

\( uplo \)

Must be 'U' or 'L' or 'F'.

If \( UPLO = 'U' \), \( a \) and \( b \) store the upper triangular parts of \( A \) and \( B \) respectively.

If \( UPLO = 'L' \), \( a \) and \( b \) store the lower triangular parts of \( A \) and \( B \) respectively.

If \( UPLO = 'F' \), \( a \) and \( b \) store the full matrices \( A \) and \( B \) respectively.

\( n \)

Sets the size of the problem. \( n > 0 \).

\( a \)

Array of dimension \( lda \) by \( n \), contains either full matrix \( A \) or upper or lower triangular part of the matrix \( A \), as specified by \( uplo \).

\( lda \)

The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

\( b \)

Array of dimension \( ldb \) by \( n \), contains either full matrix \( B \) or upper or lower triangular part of the matrix \( B \), as specified by \( uplo \).

\( ldb \)

The leading dimension of the array \( B \). Must be at least \( \max(1, n) \).

\( fpm \)

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

\( emin, emax \)

The lower and upper bounds of the interval to be searched for eigenvalues; \( emin \leq emax \).

\( m0 \)

On entry, specifies the initial guess for subspace dimension to be used, \( 0 < m0 \leq n \). Set \( m0 \geq m \) where \( m \) is the total number of eigenvalues located in the interval \([emin, emax]\). If the initial guess is wrong, Extended Eigensolver routines return \( info=3 \).

\( x \)

On entry, if \( fpm[4] = 1 \), the array \( x \) of size \( n \) by \( m \) contains a basis of guess subspace where \( n \) is the order of the input matrix.
Output Parameters

**epsout**
On output, contains the relative error on the trace: \( |\text{trace} - \text{trace}_{i-1}| / \max(|emin|, |emax|) \)

**loop**
On output, contains the number of refinement loop executed. Ignored on input.

**e**
Array of length \( m0 \). On output, the first \( m \) entries of \( e \) are eigenvalues found in the interval.

**x**
On output, the first \( m \) columns of \( x \) contain the orthonormal eigenvectors corresponding to the computed eigenvalues \( e \), with the \( i \)-th column of \( x \) holding the eigenvector associated with \( e[i] \).

**m**
The total number of eigenvalues found in the interval \( [emin, emax] \): \( 0 \leq m \leq m0 \).

**res**
Array of length \( m0 \). On exit, the first \( m \) components contain the relative residual vector:

\[
\frac{\| \Delta x_i - \hat{A} x_i \|_2}{\max(\| \bar{E}_{\text{min}} \|, \| \bar{E}_{\text{max}} \|) \| x_i \|_2}
\]

for \( i=1, 2, \ldots, m \), and where \( m \) is the total number of eigenvalues found in the search interval.

**info**
If \( info=0 \), the execution is successful. If \( info \neq 0 \), see Output Eigensolver info Details.

### ?feast_sbev/?feast_hbev

Extended Eigensolver interface for standard eigenvalue problem with banded matrices.

#### Syntax

```c
void sfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const float * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);
void dfeast_sbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const double * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT * info);
void cfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const MKL_Complex8 * a, const MKL_INT * lda, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);
void zfeast_hbev (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const MKL_Complex16 * a, const MKL_INT * lda, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);
```
Include Files

- mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, \( Ax = \lambda x \), within a given search interval.

Input Parameters

\( uplo \)

Must be 'U' or 'L' or 'F'.

- If \( uplo = 'U' \), \( a \) stores the upper triangular parts of \( A \).
- If \( uplo = 'L' \), \( a \) stores the lower triangular parts of \( A \).
- If \( uplo = 'F' \), \( a \) stores the full matrix \( A \).

\( n \)

Sets the size of the problem. \( n \) > 0.

\( kla \)

The number of super- or sub-diagonals within the band in \( A \) (\( kla \geq 0 \)).

\( a \)

Array of dimension \( lda \) by \( n \), contains either full matrix \( A \) or upper or lower triangular part of the matrix \( A \), as specified by \( uplo \).

\( lda \)

The leading dimension of the array \( a \). Must be at least \( \max(1, n) \).

\( fpm \)

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

\( emin, emax \)

The lower and upper bounds of the interval to be searched for eigenvalues; \( emin \leq emax \).

\( m0 \)

On entry, specifies the initial guess for subspace dimension to be used, \( 0 < m0 \leq n \). Set \( m0 \geq m \) where \( m \) is the total number of eigenvalues located in the interval \( [emin, emax] \). If the initial guess is wrong, Extended Eigensolver routines return info=3.

\( x \)

On entry, if \( fpm[4] = 1 \), the array \( x \) of size \( n \) by \( m \) contains a basis of guess subspace where \( n \) is the order of the input matrix.

Output Parameters

\( epsout \)

On output, contains the relative error on the trace: \( |trace_i - trace_{i-1}| / \max(|emin|, |emax|) \).

\( loop \)

On output, contains the number of refinement loop executed. Ignored on input.

\( e \)

Array of length \( m0 \). On output, the first \( m \) entries of \( e \) are eigenvalues found in the interval.

\( x \)

On output, the first \( m \) columns of \( x \) contain the orthonormal eigenvectors corresponding to the computed eigenvalues \( e_i \) with the \( i \)-th column of \( x \) holding the eigenvector associated with \( e_i \).

\( m \)

The total number of eigenvalues found in the interval \( [emin, emax] \): \( 0 \leq m \leq m0 \).
res

Array of length \( m_0 \). On exit, the first \( m \) components contain the relative residual vector:

\[
\frac{\| A \hat{x}_i - \lambda \hat{x}_i \|}{\max(\| E \|, \| \hat{x}_i \|)}
\]

for \( i = 1, 2, \ldots, m \), and where \( m \) is the total number of eigenvalues found in the search interval.

info

If \( info = 0 \), the execution is successful. If \( info \neq 0 \), see Output Eigensolver info Details.

?feast_sbgv/?feast_hbgv

Extended Eigensolver interface for generalized eigenvalue problem with banded matrices.

Syntax

void sfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const float * a, const MKL_INT * lda, const MKL_INT * klb, const float * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void dfeast_sbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const double * a, const MKL_INT * lda, const MKL_INT * klb, const double * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, double * x, MKL_INT * m, double * res, MKL_INT * info);

void cfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const MKL_Complex8 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex8 * b, const MKL_INT * ldb, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, MKL_Complex8 * x, MKL_INT * m, float * res, MKL_INT * info);

void zfeast_hbgv (const char * uplo, const MKL_INT * n, const MKL_INT * kla, const MKL_Complex16 * a, const MKL_INT * lda, const MKL_INT * klb, const MKL_Complex16 * b, const MKL_INT * ldb, MKL_INT * fpm, double * epsout, MKL_INT * loop, const double * emin, const double * emax, MKL_INT * m0, double * e, MKL_Complex16 * x, MKL_INT * m, double * res, MKL_INT * info);

Include Files

• mkl.h

Description

The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, \( Ax = \lambda Bx \), within a given search interval.

NOTE

Both matrices \( A \) and \( B \) must use the same family of storage format. The bandwidth, however, can be different (\( klb \) can be different from \( kla \)).
**Input Parameters**

**uplo**
Must be 'U' or 'L' or 'F'.

If `UPLO = 'U'`, `a` and `b` store the upper triangular parts of `A` and `B` respectively.

If `UPLO = 'L'`, `a` and `b` store the lower triangular parts of `A` and `B` respectively.

If `UPLO = 'F'`, `a` and `b` store the full matrices `A` and `B` respectively.

**n**
Sets the size of the problem. `n > 0`.

**kla**
The number of super- or sub-diagonals within the band in `A` (`kla ≥ 0`).

**a**
Array of dimension `lda` by `n`, contains either full matrix `A` or upper or lower triangular part of the matrix `A`, as specified by `uplo`.

**lda**
The leading dimension of the array `a`. Must be at least `max(1, n)`.

**klb**
The number of super- or sub-diagonals within the band in `B` (`klb ≥ 0`).

**b**
Array of dimension `ldb` by `n`, contains either full matrix `B` or upper or lower triangular part of the matrix `B`, as specified by `uplo`.

**ldb**
The leading dimension of the array `b`. Must be at least `max(1, n)`.

**fpm**
Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See [Extended Eigensolver Input Parameters](#) for a complete description of the parameters and their default values.

**emin, emax**
The lower and upper bounds of the interval to be searched for eigenvalues; `emin ≤ emax`.

**m0**
On entry, specifies the initial guess for subspace dimension to be used, `0 < m0 ≤ n`. Set `m0 ≥ m` where `m` is the total number of eigenvalues located in the interval `[emin, emax]`. If the initial guess is wrong, Extended Eigensolver routines return `info = 3`.

**x**
On entry, if `fpm[4] = 1`, the array `x` of size `n` by `m` contains a basis of guess subspace where `n` is the order of the input matrix.

**Output Parameters**

**epsout**
On output, contains the relative error on the trace: `|trace_i - trace_{i-1}| / max(|emin|, |emax|)`

**loop**
On output, contains the number of refinement loop executed. Ignored on input.

**e**
Array of length `m0`. On output, the first `m` entries of `e` are eigenvalues found in the interval.

**x**
On output, the first `m` columns of `x` contain the orthonormal eigenvectors corresponding to the computed eigenvalues `e`, with the `i`-th column of `x` holding the eigenvector associated with `e[i]`. 
The total number of eigenvalues found in the interval \([emin, emax]\): \(0 \leq m \leq m0\).

Array of length \(m0\). On exit, the first \(m\) components contain the relative residual vector:

\[
\frac{\|A\vec{x}_i - \lambda_i \vec{x}_i\|}{\max(\|\vec{x}_i\|, \|\vec{x}_i\|)}
\]

for \(i=1, 2, \ldots, m\), and where \(m\) is the total number of eigenvalues found in the search interval.

If \(info=0\), the execution is successful. If \(info \neq 0\), see Output Eigensolver info Details.

**feast_scsrev/feast_hcsrev**

Extended Eigensolver interface for standard eigenvalue problem with sparse matrices.

### Syntax

void feast_scsrev (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

void feast_hcsrev (const char * uplo, const MKL_INT * n, const float * a, const MKL_INT * ia, const MKL_INT * ja, MKL_INT * fpm, float * epsout, MKL_INT * loop, const float * emin, const float * emax, MKL_INT * m0, float * e, float * x, MKL_INT * m, float * res, MKL_INT * info);

### Include Files

- mkl.h

### Description

The routines compute all the eigenvalues and eigenvectors for standard eigenvalue problems, \(Ax = \lambda x\), within a given search interval.

### Input Parameters

**uplo**

Must be 'U' or 'L' or 'F'.

If \(uplo = 'U'\), \(a\) stores the upper triangular parts of \(A\).

If \(uplo = 'L'\), \(a\) stores the lower triangular parts of \(A\).
If uplo = 'F', a stores the full matrix A.

Sets the size of the problem. n > 0.

Array containing the nonzero elements of either the full matrix A or the upper or lower triangular part of the matrix A, as specified by uplo.

Array of length n + 1, containing indices of elements in the array a, such that ia[i] is the index in the array a of the first non-zero element from the row i. The value of the last element ia[n] is equal to the number of non-zeros plus one.

Array containing the column indices for each non-zero element of the matrix A being represented in the array a. Its length is equal to the length of the array a.

Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

The lower and upper bounds of the interval to be searched for eigenvalues; emin ≤ emax.

On entry, specifies the initial guess for subspace dimension to be used, 0 < m0 ≤ n. Set m0 ≥ m where m is the total number of eigenvalues located in the interval [emin, emax]. If the initial guess is wrong, Extended Eigensolver routines return info=3.

On entry, if fpm[4] = 1, the array x of size n by m contains a basis of guess subspace where n is the order of the input matrix.

Output Parameters

On output, the last 64 values correspond to Intel® MKL PARDISO iparm[0] to iparm[63] (regardless of the value of fpm[63] on input).

On output, contains the relative error on the trace: |trace_i - trace_{i-1}| / max(|emin|, |emax|)

On output, contains the number of refinement loop executed. Ignored on input.

Array of length m0. On output, the first m entries of e are eigenvalues found in the interval.

On output, the first m columns of x contain the orthonormal eigenvectors corresponding to the computed eigenvalues e, with the i-th column of x holding the eigenvector associated with e[i].

The total number of eigenvalues found in the interval [emin, emax]: 0 ≤ m ≤ m0.

Array of length m0. On exit, the first m components contain the relative residual vector:
\[
\frac{\|A \xi_i - \lambda_i x_i\|_k}{\max(\|A\|_\infty, \|B\|_\infty)}
\]
for \(i=1, 2, \ldots, m\), and where \(m\) is the total number of eigenvalues found in the search interval.

\[\text{info}\]

If \(\text{info}=0\), the execution is successful. If \(\text{info} \neq 0\), see Output Eigensolver info Details.

**feast_scsrgv/feast_hcsrgv**

Extended Eigensolver interface for generalized eigenvalue problem with sparse matrices.

### Syntax

```c
void sfeast_scsrgv (const char *uplo, const MKL_INT *n, const float *a, const MKL_INT *ia, const MKL_INT *ja, const float *b, const MKL_INT *ib, const MKL_INT *jb, MKL_INT *fpm, float *epsout, MKL_INT *loop, const float *emin, const float *emax, MKL_INT *m0, float *e, float *x, MKL_INT *m, float *res, MKL_INT *info);
void dfeast_scsrgv (const char *uplo, const MKL_INT *n, const double *a, const MKL_INT *ia, const MKL_INT *ja, const double *b, const MKL_INT *ib, const MKL_INT *jb, MKL_INT *fpm, double *epsout, MKL_INT *loop, const double *emin, const double *emax, MKL_INT *m0, double *e, double *x, MKL_INT *m, double *res, MKL_INT *info);
void cfeast_hcsrgv (const char *uplo, const MKL_INT *n, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, MKL_INT *fpm, float *epsout, MKL_INT *loop, const float *emin, const float *emax, MKL_INT *m0, float *e, MKL_Complex8 *x, MKL_INT *m, float *res, MKL_INT *info);
void zfeast_hcsrgv (const char *uplo, const MKL_INT *n, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, MKL_INT *fpm, double *epsout, MKL_INT *loop, const double *emin, const double *emax, MKL_INT *m0, double *e, MKL_Complex16 *x, MKL_INT *m, double *res, MKL_INT *info);
```

### Include Files
- mkl.h

### Description
The routines compute all the eigenvalues and eigenvectors for generalized eigenvalue problems, \(Ax = \lambda Bx\), within a given search interval.

**NOTE**
Both matrices \(A\) and \(B\) must use the same family of storage format. The position of the non-zero elements can be different (CSR coordinates \(ib\) and \(jb\) can be different from \(ia\) and \(ja\)).

### Input Parameters
- **uplo** Must be 'U' or 'L' or 'F'.
If `UPLO = 'U'`, `a` and `b` store the upper triangular parts of `A` and `B` respectively.

If `UPLO = 'L'`, `a` and `b` store the lower triangular parts of `A` and `B` respectively.

If `UPLO = 'F'`, `a` and `b` store the full matrices `A` and `B` respectively.

`n` 
Sets the size of the problem. `n > 0`.

`a` 
Array containing the nonzero elements of either the full matrix `A` or the upper or lower triangular part of the matrix `A`, as specified by `uplo`.

`ia` 
Array of length `n + 1`, containing indices of elements in the array `a`, such that `ia[i - 1]` is the index in the array `a` of the first non-zero element from the row `i`. The value of the last element `ia[n]` is equal to the number of non-zeros plus one.

`ja` 
Array containing the column indices for each non-zero element of the matrix `A` being represented in the array `a`. Its length is equal to the length of the array `a`.

`b` 
Array of dimension `ldb` by *, contains the nonzero elements of either the full matrix `B` or the upper or lower triangular part of the matrix `B`, as specified by `uplo`.

`ib` 
Array of length `n + 1`, containing indices of elements in the array `b`, such that `ib[i - 1]` is the index in the array `b` of the first non-zero element from the row `i`. The value of the last element `ib[n]` is equal to the number of non-zeros plus one.

`jb` 
Array containing the column indices for each non-zero element of the matrix `B` being represented in the array `b`. Its length is equal to the length of the array `b`.

`fpm` 
Array, dimension of 128. This array is used to pass various parameters to Extended Eigensolver routines. See Extended Eigensolver Input Parameters for a complete description of the parameters and their default values.

`emin, emax` 
The lower and upper bounds of the interval to be searched for eigenvalues; `emin ≤ emax`.

`m0` 
On entry, specifies the initial guess for subspace dimension to be used, `0 < m0 ≤ n`. Set `m0 ≥ m` where `m` is the total number of eigenvalues located in the interval `[emin, emax]`. If the initial guess is wrong, Extended Eigensolver routines return `info=3`.

`x` 
On entry, if `fpm[4]=1`, the array `x` of size `n` by `m` contains a basis of guess subspace where `n` is the order of the input matrix.

**Output Parameters**

`fpm` 
On output, the last 64 values correspond to Intel® MKL PARDISO `iparm[0]` to `iparm[63]` (regardless of the value of `fpm[63]` on input).

`epsout` 
On output, contains the relative error on the trace: `|trace - trace_{i-1}| / max(|emin|, |emax|)`
loop
On output, contains the number of refinement loop executed. Ignored on input.

$e$
Array of length $m0$. On output, the first $m$ entries of $e$ are eigenvalues found in the interval.

$x$
On output, the first $m$ columns of $x$ contain the orthonormal eigenvectors corresponding to the computed eigenvalues $e$, with the $i$-th column of $x$ holding the eigenvector associated with $e_i$.

$m$
The total number of eigenvalues found in the interval $[emin, emax]$: $0 \leq m \leq m0$.

$res$
Array of length $m0$. On exit, the first $m$ components contain the relative residual vector:

$$
\frac{\|Ax_i - \lambda_i Bx_i\|_1}{\max(\|E_{min}\|_1, \|E_{max}\|_1) \|Bx_i\|_1}
$$

for $i=1, 2, ..., m$, and where $m$ is the total number of eigenvalues found in the search interval.

$info$
If $info=0$, the execution is successful. If $info \neq 0$, see Output Eigensolver info Details.
Vector Mathematical Functions

This chapter describes Vector Mathematics functions (VM), which compute a mathematical function of each of the vector elements. VM includes a set of highly optimized functions (arithmetic, power, trigonometric, exponential, hyperbolic, special, and rounding) that operate on vectors of real and complex numbers.

Application programs that improve performance with VM include nonlinear programming software, computation of integrals, financial calculations, computer graphics, and many others.

VM functions fall into the following groups according to the operations they perform:

- **VM Mathematical Functions** compute values of mathematical functions, such as sine, cosine, exponential, or logarithm, on vectors stored contiguously in memory.
- **VM Pack/Unpack Functions** convert to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix B for details on vector indexing methods).
- **VM Service Functions** set/get the accuracy modes and the error codes, and free memory.

The VM mathematical functions take an input vector as an argument, compute values of the respective function element-wise, and return the results in an output vector. All the VM mathematical functions can perform in-place operations, where the input and output arrays are at the same memory locations.

The interfaces are given in `<mkl_vml_functions.h>`.

Examples that demonstrate how to use the VM functions are located in: `${MKL}/examples/vmlc/source`


VM Data Types, Accuracy Modes, and Performance Tips

VM includes mathematical and pack/unpack vector functions for single and double precision vector arguments of real and complex types. Intel® MKL provides Fortran and C interfaces for all VM functions, including the associated service functions. The Function Naming Conventions section below shows how to call these functions.

Performance depends on a number of factors, including vectorization and threading overhead. The recommended usage is as follows:

- Use VM for vector lengths larger than 40 elements.
- Use the Intel® Compiler for vector lengths less than 40 elements.

All VM vector functions support the following accuracy modes:

- **High Accuracy (HA)**, the default mode
- **Low Accuracy (LA)**, which improves performance by reducing accuracy of the two least significant bits
- **Enhanced Performance (EP)**, which provides better performance at the cost of significantly reduced accuracy. Approximately half of the bits in the mantissa are correct.

Note that using the EP mode does not guarantee accurate processing of corner cases and special values. Although the default accuracy is HA, LA is sufficient in most cases. For applications that require less accuracy (for example, media applications, some Monte Carlo simulations, etc.), the EP mode may be sufficient.

VM handles special values in accordance with the C99 standard [C99].

Intel® MKL offers both functions and environment variables to switch between modes for VM. See the Intel® MKL Developer Guide for details about the environment variables. Use the `vmlSetMode(mode)` function (see Table "Values of the mode Parameter") to switch between the HA, LA, and EP modes. The `vmlGetMode()` function returns the current mode.
### Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### See Also

- Function Naming Conventions

### VM Naming Conventions

The VM function names are of mixed (lower and upper) case.

The VM mathematical and pack/unpack function names have the following structure:

```
v[m]<?><name><mod>
```

where

- `v` is a prefix indicating vector operations.
- `[m]` is an optional prefix for mathematical functions that indicates additional argument to specify a VM mode for a given function call (see `vmlSetMode` for possible values and their description).
- `<?>` is a precision prefix that indicates one of the following the data types:
  - `s` float.
  - `d` double.
  - `c` MKL_Complex8.
  - `z` MKL_Complex16.
- `<name>` indicates the function short name, with some of its letters in uppercase. See examples in Table "VM Mathematical Functions".
- `<mod>` field (written in uppercase) is present only in the pack/unpack functions and indicates the indexing method used:
  - `i` indexing with a positive increment
  - `v` indexing with an index vector
  - `m` indexing with a mask vector.

The VM service function names have the following structure:

```
vm/<name>
```

where

- `<name>` indicates the function short name, with some of its letters in uppercase. See examples in Table "VM Service Functions".

To call VM functions from an application program, use conventional function calls. For example, call the vector single precision real exponential function as

```
vsExp ( n, a, y );
```
VM Function Interfaces

VM interfaces include the function names and argument lists. The following sections describe the interfaces for the VM functions. Note that some of the functions have multiple input and output arguments.

Some VM functions may also take scalar arguments as input. See the function description for the naming conventions of such arguments.

VM Mathematical Function Interfaces

\[ v\langle ?\rangle <name>( n, a, \{scalar input arguments,\}, y ); \]

\[ v\langle ?\rangle <name>( n, a, b, \{scalar input arguments,\}, y ); \]

\[ vm\langle ?\rangle <name>( n, a, y, z ); \]

\[ vm\langle ?\rangle <name>( n, a, \{scalar input arguments,\}, y, mode ); \]

\[ vm\langle ?\rangle <name>( n, a, b, \{scalar input arguments,\}, y, mode ); \]

\[ vm\langle ?\rangle <name>( n, a, y, z, mode ); \]

VM Pack Function Interfaces

\[ v\langle ?\rangle PackI( n, a, inca, y ); \]

\[ v\langle ?\rangle PackV( n, a, ia, y ); \]

\[ v\langle ?\rangle PackM( n, a, ma, y ); \]

VM Unpack Function Interfaces

\[ v\langle ?\rangle UnpackI( n, a, y, incy ); \]

\[ v\langle ?\rangle UnpackV( n, a, y, iy ); \]

\[ v\langle ?\rangle UnpackM( n, a, y, my ); \]

VM Service Function Interfaces

\[ oldmode = vmlSetMode( mode ); \]

\[ mode = vmlGetMode( void ); \]

\[ olderr = vmlSetErrStatus( err ); \]

\[ err = vmlGetErrStatus( void ); \]

\[ olderr = vmlClearErrStatus( void ); \]

\[ oldcallback = vmlSetErrorCallBack( callback ); \]

\[ callback = vmlGetErrorCallBack( void ); \]

\[ oldcallback = vmlClearErrorCallBack( void ); \]

Note that oldmode, oldcerr, and oldcallback refer to settings prior to the call.

VM Input Function Interfaces

\[ n \quad \text{number of elements to be calculated} \]

\[ a \quad \text{first input vector} \]

\[ b \quad \text{second input vector} \]

\[ inca \quad \text{vector increment for the input vector } a \]

\[ ia \quad \text{index vector for the input vector } a \]

\[ ma \quad \text{mask vector for the input vector } a \]

\[ incy \quad \text{vector increment for the output vector } y \]

\[ iy \quad \text{index vector for the output vector } y \]
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>my</td>
<td>mask vector for the output vector y</td>
</tr>
<tr>
<td>err</td>
<td>error code</td>
</tr>
<tr>
<td>mode</td>
<td>VM mode</td>
</tr>
<tr>
<td>callback</td>
<td>address of the callback function</td>
</tr>
</tbody>
</table>

**VM Output Function Interfaces**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>first output vector</td>
</tr>
<tr>
<td>z</td>
<td>second output vector</td>
</tr>
<tr>
<td>err</td>
<td>error code</td>
</tr>
<tr>
<td>mode</td>
<td>VM mode</td>
</tr>
<tr>
<td>olderr</td>
<td>former error code</td>
</tr>
<tr>
<td>oldmode</td>
<td>former VM mode</td>
</tr>
<tr>
<td>callback</td>
<td>address of the callback function</td>
</tr>
<tr>
<td>oldcallback</td>
<td>address of the former callback function</td>
</tr>
</tbody>
</table>

See the data types of the parameters used in each function in the respective function description section. All the Intel® MKL VM mathematical functions can perform in-place operations.

**Vector Indexing Methods**

VM mathematical functions work only with unit stride. To accommodate arrays with other increments, or more complicated indexing, you can gather the elements into a contiguous vector and then scatter them after the computation is complete.

VM uses the three following indexing methods to do this task:

- positive increment
- index vector
- mask vector

The indexing method used in a particular function is indicated by the indexing modifier (see the description of the `<mod>` field in Function Naming Conventions). For more information on the indexing methods, see Vector Arguments in VM in Appendix B.

**VM Error Diagnostics**

The VM mathematical functions incorporate the error handling mechanism, which is controlled by the following service functions:

- `vmlGetErrStatus`, `vmlSetErrStatus`, `vmlClearErrStatus`
  These functions operate with a global variable called VM Error Status. The VM Error Status flags an error, a warning, or a successful execution of a VM function.

- `vmlGetErrCallBack`, `vmlSetErrCallBack`, `vmlClearErrCallBack`
  These functions enable you to customize the error handling. For example, you can identify a particular argument in a vector where an error occurred or that caused a warning.
These functions get and set a VM mode. If you set a new VM mode using the vmlSetMode function, you can store the previous VM mode returned by the routine and restore it at any point of your application.

If both an error and a warning situation occur during the function call, the VM Error Status variable keeps only the value of the error code. See Table "Values of the VM Error Status" for possible values. If a VM function does not encounter errors or warnings, it sets the VM Error Status to VML_STATUS_OK.

If you use incorrect input arguments to a VM function (VML_STATUS_BADSIZE and VML_STATUS_BADMEM), the function calls xerbla to report errors. See Table "Values of the VM Error Status" for details.

You can use the vmlSetMode and vmlGetMode functions to modify error handling behavior. Depending on the VM mode, the error handling behavior includes the following operations:

- setting the VM Error Status to a value corresponding to the observed error or warning
- setting the errno variable to one of the values described in Table "Set Values of the errno Variable"
- writing error text information to the stderr stream
- raising the appropriate exception on an error, if necessary
- calling the additional error handler callback function that is set by vmlSetErrorCallBack.

<table>
<thead>
<tr>
<th>Value of errno</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No errors are detected.</td>
</tr>
<tr>
<td>EINVAL</td>
<td>The array dimension is not positive.</td>
</tr>
<tr>
<td>EACCES</td>
<td>NULL pointer is passed.</td>
</tr>
<tr>
<td>EDOM</td>
<td>At least one of array values is out of a range of definition.</td>
</tr>
<tr>
<td>ERANGE</td>
<td>At least one of array values caused a singularity, overflow or underflow.</td>
</tr>
</tbody>
</table>

See Also

- vmlGetErrStatus Gets the VM Error Status.
- vmlSetErrStatus Sets the new VM Error Status according to err and stores the previous VM Error Status to olderr.
- vmlClearErrStatus Sets the VM Error Status to VML_STATUS_OK and stores the previous VM Error Status to olderr.
- vmlSetErrorCallBack Sets the additional error handler callback function and gets the old callback function.
- vmlGetErrorCallBack Gets the additional error handler callback function.
- vmlClearErrorCallBack Deletes the additional error handler callback function and retrieves the former callback function.
- vmlGetMode Gets the VM mode.
- vmlSetMode Sets a new mode for VM functions according to the mode parameter and stores the previous VM mode to oldmode.

**VM Mathematical Functions**

This section describes VM functions that compute values of mathematical functions on real and complex vector arguments with unit increment.

Each function is introduced by its short name, a brief description of its purpose, and the calling sequence for each type of data, as well as a description of the input/output arguments.

The input range of parameters is equal to the mathematical range of the input data type, unless the function description specifies input threshold values, which mark off the precision overflow, as follows:
FLT_MAX denotes the maximum number representable in single precision real data type

DBL_MAX denotes the maximum number representable in double precision real data type

Table "VM Mathematical Functions" lists available mathematical functions and associated data types.

<table>
<thead>
<tr>
<th>Function</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Add</td>
<td>s, d, c, z</td>
<td>Adds vector elements</td>
</tr>
<tr>
<td>v?Sub</td>
<td>s, d, c, z</td>
<td>Subtracts vector elements</td>
</tr>
<tr>
<td>v?Sqr</td>
<td>s, d</td>
<td>Squares vector elements</td>
</tr>
<tr>
<td>v?Mul</td>
<td>s, d, c, z</td>
<td>Multiplies vector elements</td>
</tr>
<tr>
<td>v?MulByConj</td>
<td>c, z</td>
<td>Multiplies elements of one vector by conjugated elements of the second vector</td>
</tr>
<tr>
<td>v?Conj</td>
<td>c, z</td>
<td>Conjugates vector elements</td>
</tr>
<tr>
<td>v?Abs</td>
<td>s, d, c, z</td>
<td>Computes the absolute value of vector elements</td>
</tr>
<tr>
<td>v?Arg</td>
<td>c, z</td>
<td>Computes the argument of vector elements</td>
</tr>
<tr>
<td>v?LinearFrac</td>
<td>s, d</td>
<td>Performs linear fraction transformation of vectors</td>
</tr>
<tr>
<td>v?Fmod</td>
<td>s, d</td>
<td>Performs element by element computation of the modulus function of vector a with respect to vector b</td>
</tr>
<tr>
<td>v?Remainder</td>
<td>s, d</td>
<td>Performs element by element computation of the remainder function on the elements of vector a and the corresponding elements of vector b</td>
</tr>
<tr>
<td><strong>Power and Root Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Inv</td>
<td>s, d</td>
<td>Inverts vector elements</td>
</tr>
<tr>
<td>v?Div</td>
<td>s, d, c, z</td>
<td>Divides elements of one vector by elements of the second vector</td>
</tr>
<tr>
<td>v?Sqrt</td>
<td>s, d, c, z</td>
<td>Computes the square root of vector elements</td>
</tr>
<tr>
<td>v?InvSqrt</td>
<td>s, d</td>
<td>Computes the inverse square root of vector elements</td>
</tr>
<tr>
<td>v?Cbrt</td>
<td>s, d</td>
<td>Computes the cube root of vector elements</td>
</tr>
<tr>
<td>v?InvCbrt</td>
<td>s, d</td>
<td>Computes the inverse cube root of vector elements</td>
</tr>
<tr>
<td>v?Pow2o3</td>
<td>s, d</td>
<td>Computes the cube root of the square of each vector element</td>
</tr>
<tr>
<td>v?Pow3o2</td>
<td>s, d</td>
<td>Computes the square root of the cube of each vector element</td>
</tr>
<tr>
<td>v?Pow</td>
<td>s, d, c, z</td>
<td>Raises each vector element to the specified power</td>
</tr>
<tr>
<td>v?Powr</td>
<td>s, d, c, z</td>
<td>Computes a to the power b for elements of two vectors, where the elements of vector argument a are all non-negative</td>
</tr>
<tr>
<td>v?Hypot</td>
<td>s, d</td>
<td>Computes the square root of sum of squares</td>
</tr>
<tr>
<td><strong>Exponential and Logarithmic Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Exp</td>
<td>s, d, c, z</td>
<td>Computes the base e exponential of vector elements</td>
</tr>
<tr>
<td>v?Exp2</td>
<td>s, d</td>
<td>Computes the base 2 exponential of vector elements</td>
</tr>
<tr>
<td>v?Exp10</td>
<td>s, d</td>
<td>Computes the base 10 exponential of vector elements</td>
</tr>
<tr>
<td>v?Expm1</td>
<td>s, d</td>
<td>Computes the base e exponential of vector elements decreased by 1</td>
</tr>
<tr>
<td>v?Ln</td>
<td>s, d, c, z</td>
<td>Computes the natural logarithm of vector elements</td>
</tr>
<tr>
<td>v?Log2</td>
<td>s, d</td>
<td>Computes the base 2 logarithm of vector elements</td>
</tr>
<tr>
<td>v?Log10</td>
<td>s, d, c, z</td>
<td>Computes the base 10 logarithm of vector elements</td>
</tr>
<tr>
<td>v?Log1p</td>
<td>s, d</td>
<td>Computes the natural logarithm of vector elements that are increased by 1</td>
</tr>
<tr>
<td>v?Logb</td>
<td>s, d</td>
<td>Computes the exponents of the elements of input vector a</td>
</tr>
<tr>
<td><strong>Trigonometric Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Cos</td>
<td>s, d, c, z</td>
<td>Computes the cosine of vector elements</td>
</tr>
<tr>
<td>v?Sin</td>
<td>s, d, c, z</td>
<td>Computes the sine of vector elements</td>
</tr>
<tr>
<td>v?SinCos</td>
<td>s, d</td>
<td>Computes the sine and cosine of vector elements</td>
</tr>
<tr>
<td>v?CIS</td>
<td>c, z</td>
<td>Computes the complex exponent of vector elements (cosine and sine combined to complex value)</td>
</tr>
<tr>
<td>v?Tan</td>
<td>s, d, c, z</td>
<td>Computes the tangent of vector elements</td>
</tr>
</tbody>
</table>

Vector Mathematical Functions
<table>
<thead>
<tr>
<th>Function</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v?Acos</td>
<td>s, d, c, z</td>
<td>Computes the inverse cosine of vector elements</td>
</tr>
<tr>
<td>v?Asin</td>
<td>s, d, c, z</td>
<td>Computes the inverse sine of vector elements</td>
</tr>
<tr>
<td>v?Atan</td>
<td>s, d, c, z</td>
<td>Computes the inverse tangent of vector elements</td>
</tr>
<tr>
<td>v?Atan2</td>
<td>s, d</td>
<td>Computes the four-quadrant inverse tangent of ratios of the elements of two vectors</td>
</tr>
<tr>
<td>v?Cospi</td>
<td>s, d</td>
<td>Computes the cosine of vector elements multiplied by ( n )</td>
</tr>
<tr>
<td>v?Sinpi</td>
<td>s, d</td>
<td>Computes the sine of vector elements multiplied by ( n )</td>
</tr>
<tr>
<td>v?Tanpi</td>
<td>s, d</td>
<td>Computes the tangent of vector elements multiplied by ( n )</td>
</tr>
<tr>
<td>v?Acospri</td>
<td>s, d</td>
<td>Computes the inverse cosine of vector elements divided by ( n )</td>
</tr>
<tr>
<td>v?Asinpi</td>
<td>s, d</td>
<td>Computes the inverse sine of vector elements divided by ( n )</td>
</tr>
<tr>
<td>v?Atanpi</td>
<td>s, d</td>
<td>Computes the inverse tangent of vector elements divided by ( n )</td>
</tr>
<tr>
<td>v?Atan2pi</td>
<td>s, d</td>
<td>Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by ( n )</td>
</tr>
<tr>
<td>v?Cosd</td>
<td>s, d</td>
<td>Computes the cosine of vector elements multiplied by ( n/180 )</td>
</tr>
<tr>
<td>v?Sind</td>
<td>s, d</td>
<td>Computes the sine of vector elements multiplied by ( n/180 )</td>
</tr>
<tr>
<td>v?Tand</td>
<td>s, d</td>
<td>Computes the tangent of vector elements multiplied by ( n/180 )</td>
</tr>
<tr>
<td><strong>Hyperbolic Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Cosh</td>
<td>s, d, c, z</td>
<td>Computes the hyperbolic cosine of vector elements</td>
</tr>
<tr>
<td>v?Sinh</td>
<td>s, d, c, z</td>
<td>Computes the hyperbolic sine of vector elements</td>
</tr>
<tr>
<td>v?Tanh</td>
<td>s, d, c, z</td>
<td>Computes the hyperbolic tangent of vector elements</td>
</tr>
<tr>
<td>v?Acosh</td>
<td>s, d, c, z</td>
<td>Computes the inverse hyperbolic cosine of vector elements</td>
</tr>
<tr>
<td>v?Asinh</td>
<td>s, d, c, z</td>
<td>Computes the inverse hyperbolic sine of vector elements</td>
</tr>
<tr>
<td>v?Atanh</td>
<td>s, d, c, z</td>
<td>Computes the inverse hyperbolic tangent of vector elements</td>
</tr>
<tr>
<td><strong>Special Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Erf</td>
<td>s, d</td>
<td>Computes the error function value of vector elements</td>
</tr>
<tr>
<td>v?Erfc</td>
<td>s, d</td>
<td>Computes the complementary error function value of vector elements</td>
</tr>
<tr>
<td>v?CdfNorm</td>
<td>s, d</td>
<td>Computes the cumulative normal distribution function value of vector elements</td>
</tr>
<tr>
<td>v?ErfInv</td>
<td>s, d</td>
<td>Computes the inverse error function value of vector elements</td>
</tr>
<tr>
<td>v?ErfcInv</td>
<td>s, d</td>
<td>Computes the inverse complementary error function value of vector elements</td>
</tr>
<tr>
<td>v?CdfNormInv</td>
<td>s, d</td>
<td>Computes the inverse cumulative normal distribution function value of vector elements</td>
</tr>
<tr>
<td>v?LGammav?LGamma</td>
<td>s, d</td>
<td>Computes the natural logarithm for the absolute value of the gamma function of vector elements</td>
</tr>
<tr>
<td>v?TGamma</td>
<td>s, d</td>
<td>Computes the gamma function of vector elements</td>
</tr>
<tr>
<td>v?ExpInt1</td>
<td>s, d</td>
<td>Computes the exponential integral of vector elements</td>
</tr>
<tr>
<td><strong>Rounding Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?Floor</td>
<td>s, d</td>
<td>Rounds towards minus infinity</td>
</tr>
<tr>
<td>v?Ceil</td>
<td>s, d</td>
<td>Rounds towards plus infinity</td>
</tr>
<tr>
<td>v?Trunc</td>
<td>s, d</td>
<td>Rounds towards zero infinity</td>
</tr>
<tr>
<td>v?Round</td>
<td>s, d</td>
<td>Rounds to nearest integer</td>
</tr>
<tr>
<td>v?NearbyInt</td>
<td>s, d</td>
<td>Rounds according to current mode</td>
</tr>
<tr>
<td>v?Rint</td>
<td>s, d</td>
<td>Rounds according to current mode and raising inexact result exception</td>
</tr>
<tr>
<td>v?Modf</td>
<td>s, d</td>
<td>Computes the integer and fractional parts</td>
</tr>
<tr>
<td>v?Frac</td>
<td>s, d</td>
<td>Computes the fractional part</td>
</tr>
<tr>
<td><strong>Miscellaneous Functions</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v?CopySign</td>
<td>s, d</td>
<td>Returns vector of elements of one argument with signs changed to match other argument elements</td>
</tr>
</tbody>
</table>
### Function Descriptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v?NextAfter</td>
<td><code>s, d</code></td>
<td>Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.</td>
</tr>
<tr>
<td>v?Fdim</td>
<td><code>s, d</code></td>
<td>Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.</td>
</tr>
<tr>
<td>v?Fmax</td>
<td><code>s, d</code></td>
<td>Returns the larger of each pair of elements of the two vector arguments.</td>
</tr>
<tr>
<td>v?Fmin</td>
<td><code>s, d</code></td>
<td>Returns the smaller of each pair of elements of the two vector arguments.</td>
</tr>
<tr>
<td>v?MaxMag</td>
<td><code>s, d</code></td>
<td>Returns the element with the larger magnitude between each pair of elements of the two vector arguments.</td>
</tr>
<tr>
<td>v?MinMag</td>
<td><code>s, d</code></td>
<td>Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.</td>
</tr>
</tbody>
</table>

### Special Value Notations

This section defines notations of special values for complex functions. The definitions are provided in text, tables, or formulas.

- `z, z1, z2, x, x1, x2, y, y1, y2` denote complex numbers.
- `i, i^2=-1` is the imaginary unit.
- `X, Y, x, y, x1, x2, y1, y2` denote real or imaginary parts.
- `X` and `Y` represent any finite positive IEEE-754 floating point values, if not stated otherwise.
- Quiet NaN and signaling NaN are denoted with QNAN and SNAN, respectively.
- The IEEE-754 positive infinities or floating-point numbers are denoted with a + sign before `X, Y`, etc.
- The IEEE-754 negative infinities or floating-point numbers are denoted with a − sign before `X, Y`, etc.

**CONJ(z)** and **CIS(z)** are defined as follows:

- **CONJ(x+i·y)=x−i·y**
- **CIS(y)=cos(y)+i·sin(y).**

The special value tables show the result of the function for the `z` argument at the intersection of the **RE(z)** column and the `i·IM(z)` row. If the function raises an exception on the argument `z`, the lower part of this cell shows the raised exception and the VM Error Status. An empty cell indicates that this argument is normal and the result is defined mathematically.

### Arithmetic Functions

Arithmetic functions perform the basic mathematical operations like addition, subtraction, multiplication or computation of the absolute value of the vector elements.

**v?Add**  
*Performs element by element addition of vector a and vector b.*

**Syntax**

```c
vsAdd( n, a, b, y );
vmsAdd( n, a, b, y, mode );
vdAdd( n, a, b, y );
vmdAdd( n, a, b, y, mode );
vcAdd( n, a, b, y );
```
vmcAdd( n, a, b, y, mode );
vzAdd( n, a, b, y );
vmzAdd( n, a, b, y, mode );

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsAdd, vmsAdd; const double* for vdAdd, vmdAdd; const MKL_Complex8* for vcAdd, vmcAdd; const MKL_Complex16* for vzAdd, vmzAdd</td>
<td>Pointers to arrays that contain the input vectors a and b.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAdd, vmsAdd; double* for vdAdd, vmdAdd; MKL_Complex8* for vcAdd, vmcAdd; MKL_Complex16* for vzAdd, vmzAdd</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

**Description**
The v?Add function performs element by element addition of vector a and vector b.

**Special values for Real Function v?Add**

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>-∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>+∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>any value</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>any value</td>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>
Specifications for special values of the complex functions are defined according to the following formula:

\[ \text{Add}(x_1+i*y_1, x_2+i*y_2) = (x_1+x_2) + i\cdot(y_1+y_2) \]

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all \( \text{RE}(x) \), \( \text{RE}(y) \), \( \text{IM}(x) \), and \( \text{IM}(y) \) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns \( \infty \) in that part of the result, raises the \text{OVERFLOW} exception, and sets the VM Error Status to \text{VML_STATUS_OVERFLOW} (overriding any possible \text{VML_STATUS_ACCURACYWARNING} status).

**v?Sub**

Performs element by element subtraction of vector \( b \) from vector \( a \).

**Syntax**

```c
vsSub( n, a, b, y );
vmsSub( n, a, b, y, mode );
vdSub( n, a, b, y );
vmdSub( n, a, b, y, mode );
vcSub( n, a, b, y );
vmcSub( n, a, b, y, mode );
vzSub( n, a, b, y );
vmzSub( n, a, b, y, mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a, b )</td>
<td>const float* for vsSub, vmsSub</td>
<td>Pointers to arrays that contain the input vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td></td>
<td>const double* for vdSub, vmdSub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex8* for vcSub, vmcSub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16* for vzSub, vmzSub</td>
<td></td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See \text{vmlSetMode} for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsSub, vmsSub</td>
<td>Pointer to an array that contains the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>double* for vdSub, vmdSub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8* for vcSub, vmcSub</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16* for vzSub, vmzSub</td>
<td></td>
</tr>
</tbody>
</table>

Description

The $v?Sub$ function performs element by element subtraction of vector $b$ from vector $a$.

Special values for Real Function $v?Sub(x)$

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>-∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>+∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>any value</td>
<td>SNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>any value</td>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>non-SNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>non-SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula:

$$
\text{Sub}(x_1+i*y_1,x_2+i*y_2) = (x_1-x_2) + i(y_1-y_2).
$$

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x), \text{RE}(y), \text{IM}(x), \text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns $\infty$ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

$v?Sqr$

Performs element by element squaring of the vector.

Syntax

```c
vsSqr( n, a, y );
vmSqr( n, a, y, mode );
vdSqr( n, a, y );
vmdSqr( n, a, y, mode );
```

Include Files

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsSqr, vmsSqr</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdSqr, vmdSqr</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsSqr, vmsSqr</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdSqr, vmdSqr</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Sqr function performs element by element squaring of the vector.

Special Values for Real Function v?Sqr(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

v?Mul

Performs element by element multiplication of vector a and vector b.

Syntax

vsMul( n, a, b, y );
vmvsMul( n, a, b, y, mode );
vdMul( n, a, b, y );
vmdMul( n, a, b, y, mode );
vcMul( n, a, b, y );
vmcMul( n, a, b, y, mode );
vszMul( n, a, b, y );
vmzMul( n, a, b, y, mode );
Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
| a, b | const float* for vsMul, vmsMul  
     | const double* for vdMul, vmdMul  
     | const MKL_Complex8* for vcMul, vmcMul  
     | const MKL_Complex16* for vzMul, vmzMul | Pointers to arrays that contain the input vectors a and b. |
| mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description. |

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| y    | float* for vsMul, vmsMul  
     | double* for vdMul, vmdMul  
     | MKL_Complex8* for vcMul, vmcMul  
     | MKL_Complex16* for vzMul, vmzMul | Pointer to an array that contains the output vector y. |

Description

The v?Mul function performs element by element multiplication of vector a and vector b.

Special values for Real Function v?Mul(x)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>-∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-0</td>
<td>+∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+0</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>-0</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>+0</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>-0</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
</tbody>
</table>
Specifications for special values of the complex functions are defined according to the following formula
\[ \text{Mul}(x_1+i*y_1, x_2+i*y_2) = (x_1*x_2-y_1*y_2) + i*(x_1*y_2+y_1*x_2). \]

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all \(\text{RE}(x), \text{RE}(y), \text{IM}(x), \text{IM}(y)\) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns \(\infty\) in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

**v?MulByConj**

*Performs element by element multiplication of vector \(a\) element and conjugated vector \(b\) element.*

**Syntax**

- vcMulByConj( \(n, a, b, y\) );
- vmcMulByConj( \(n, a, b, y, \text{mode}\) );
- vzMulByConj( \(n, a, b, y\) );
- vmzMulByConj( \(n, a, b, y, \text{mode}\) );

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a, b)</td>
<td>const MKL_Complex8* for vcMulByConj, vmcMulByConj</td>
<td>Pointers to arrays that contain the input vectors (a) and (b).</td>
</tr>
<tr>
<td>(mode)</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>MKL_Complex8* for vcMulByConj, vmcMulByConj MKL_Complex16* for vzMulByConj, vmzMulByConj</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

**Description**

The v?MulByConj function performs element by element multiplication of vector a element and conjugated vector b element.

Specifications for special values of the functions are found according to the formula

\[
\text{MulByConj}(x_1+i*y_1,x_2+i*y_2) = \text{Mul}(x_1+i*y_1,x_2-i*y_2).
\]

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all \(\text{RE}(x), \text{RE}(y), \text{IM}(x), \text{IM}(y)\) arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns \(\infty\) in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW (overriding any possible VML_STATUS_ACCURACYWARNING status).

**v?Conj**

Performs element by element conjugation of the vector.

**Syntax**

vcConj( n, a, y );
vmcConj( n, a, y, mode );
vzConj( n, a, y );
vmzConj( n, a, y, mode );

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const MKL_Complex8* for vcConj, vmcConj MKL_Complex16* for vzConj, vmzConj</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
## Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>MKL_Complex8* for vcConj, vmcConj&lt;br&gt;MKL_Complex16* for vzConj, vmzConj</td>
<td>Pointer to an array that contains the output vector $y$.</td>
</tr>
</tbody>
</table>

## Description

The $v?$Conj function performs element by element conjugation of the vector. No special values are specified. The function does not raise floating-point exceptions.

### $v?$Abs

**Computes absolute value of vector elements.**

### Syntax

```c
vsAbs( n, a, y );
vdAbs( n, a, y );
vmdAbs( n, a, y, mode );
vcAbs( n, a, y );
vmcAbs( n, a, y, mode );
vzAbs( n, a, y );
vmzAbs( n, a, y, mode );
```

## Include Files

- mkl.h

## Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$</td>
<td>const float* for vsAbs, vmsAbs&lt;br&gt;const double* for vdAbs, vmdAbs&lt;br&gt;const MKL_Complex8* for vcAbs, vmcAbs&lt;br&gt;const MKL_Complex16* for vzAbs, vmzAbs</td>
<td>Pointer to an array that contains the input vector $a$.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float*</td>
<td>for vsAbs, vmsAbs, vcAbs, vmcAbs</td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td>for vdAbs, vmdAbs, vzAbs, vmzAbs</td>
</tr>
</tbody>
</table>

Description

The v?Abs function computes an absolute value of vector elements.

Special Values for Real Function v?Abs(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

\[ \text{Abs}(z) = \text{Hypot}(\text{RE}(z), \text{IM}(z)) \]

v?Arg

**Computes argument of vector elements.**

Syntax

```c
vcArg( n, a, y );
vmcArg( n, a, y, mode );
vzArg( n, a, y );
vmzArg( n, a, y, mode );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const MKL_Complex8*</td>
<td>for vcArg, vmcArg</td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16*</td>
<td>for vzArg, vmcArg</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vcArg, vcmArg, double* for vzArg, vcmArg</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

**Description**
The `v?Arg` function computes argument of vector elements.

See the Special Value Notations section for the conventions used in the table below.

### Special Values for Complex Function v?Arg(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+i∞</td>
<td>+3π/4</td>
<td>+π/2</td>
<td>+π/2</td>
<td>+π/2</td>
<td>+π/2</td>
<td>+π/4</td>
<td>NAN</td>
<td></td>
</tr>
<tr>
<td>+iY</td>
<td>+π</td>
<td>+π/2</td>
<td>+π/2</td>
<td>+0</td>
<td>0</td>
<td>+0</td>
<td>+0</td>
<td>NAN</td>
</tr>
<tr>
<td>+i0</td>
<td>+π</td>
<td>+π</td>
<td>+π</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>NAN</td>
</tr>
<tr>
<td>-i0</td>
<td>-π</td>
<td>-π</td>
<td>-π</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td>NAN</td>
</tr>
<tr>
<td>-iY</td>
<td>-π</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/4</td>
<td>NAN</td>
<td></td>
</tr>
<tr>
<td>-i∞</td>
<td>-3π/4</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/2</td>
<td>-π/4</td>
<td>NAN</td>
<td></td>
</tr>
<tr>
<td>+iNAN</td>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- raises INVALID exception when real or imaginary part of the argument is SNAN
- `Arg(z) = Atan2(IM(z),RE(z)).`

### v?LinearFrac

Performs linear fraction transformation of vectors a and b with scalar parameters.

**Syntax**

- `vsLinearFrac( n, a, b, scalea, shiftea, scaleb, shifteb, y );`
- `vmsLinearFrac( n, a, b, scalea, shiftea, scaleb, shifteb, y, mode );`
- `vdLinearFrac( n, a, b, scalea, shiftea, scaleb, shifteb, y );`
- `vmdLinearFrac( n, a, b, scalea, shiftea, scaleb, shifteb, y, mode );`

**Include Files**
- `mkl.h`
**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a, b )</td>
<td>const float* for vsLinearFrac, vmsLinearFrac; const double* for vdLinearFrac, vmdLinearFrac</td>
<td>Pointers to arrays that contain the input vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td>( \text{scalea}, \text{scaleb} )</td>
<td>const float for vsLinearFrac, vmsLinearFrac; const double for vdLinearFrac, vmdLinearFrac</td>
<td>Constant values for scaling multipliers of vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td>( \text{shifta}, \text{shiftb} )</td>
<td>const float for vsLinearFrac, vmsLinearFrac; const double for vdLinearFrac, vmdLinearFrac</td>
<td>Constant values for shifting addends of vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td>( \text{mode} )</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See ( \text{vmlSetMode} ) for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float* for vsLinearFrac, vmsLinearFrac; double* for vdLinearFrac, vmdLinearFrac</td>
<td>Pointer to an array that contains the output vector ( y ).</td>
</tr>
</tbody>
</table>

**Description**

The \( v?\text{LinearFrac} \) function performs a linear fraction transformation of vector \( a \) by vector \( b \) with scalar parameters: scaling multipliers \( \text{scalea}, \text{scaleb} \) and shifting addends \( \text{shifta}, \text{shiftb} \):

\[
y[i] = (\text{scalea} \cdot a[i] + \text{shifta}) / (\text{scaleb} \cdot b[i] + \text{shiftb}), \ i = 1, 2 \ldots \ n
\]

The \( v?\text{LinearFrac} \) function is implemented in the EP accuracy mode only, therefore no special values are defined for this function. If used in HA or LA mode, \( v?\text{LinearFrac} \) sets the VM Error Status to \( \text{VML_STATUS_ACCURACYWARNING} \) (see the Values of the VM Status table). Correctness is guaranteed within the threshold limitations defined for each input parameter (see the table below); otherwise, the behavior is unspecified.

**Threshold Limitations on Input Parameters**

\[
\begin{align*}
2^{E_{\text{MIN}}/2} & \leq |\text{scalea}| \leq 2^{E_{\text{MAX}}-2)/2} \\
2^{E_{\text{MIN}}/2} & \leq |\text{scaleb}| \leq 2^{E_{\text{MAX}}-2)/2} \\
|\text{shifta}| & \leq 2^{E_{\text{MAX}}-2}
\end{align*}
\]
Threshold Limitations on Input Parameters

| $|\text{shiftb}| \leq 2^{\text{E}_{\text{MAX}}-2}$ | $2^{\text{E}_{\text{MIN}}/2} \leq a[i] \leq 2^{(\text{E}_{\text{MAX}}-2)/2}$ |
| $2^{\text{E}_{\text{MIN}}/2} \leq b[i] \leq 2^{(\text{E}_{\text{MAX}}-2)/2}$ |

$a[i] \neq - (\text{shifta}/\text{scalea}) \ast (1-\delta_1), \ |\delta_1| \leq 2^{1-(p-1)/2}$

$b[i] \neq - (\text{shiftb}/\text{scaleb}) \ast (1-\delta_2), \ |\delta_2| \leq 2^{1-(p-1)/2}$

$\text{E}_{\text{MIN}}$ and $\text{E}_{\text{MAX}}$ are the minimum and maximum exponents and $p$ is the number of significant bits (precision) for the corresponding data type according to the ANSI/IEEE Standard 754-2008 ([IEEE754]):

- for single precision $\text{E}_{\text{MIN}} = -126$, $\text{E}_{\text{MAX}} = 127$, $p = 24$
- for double precision $\text{E}_{\text{MIN}} = -1022$, $\text{E}_{\text{MAX}} = 1023$, $p = 53$

The thresholds become less strict for common cases with $\text{scalea}=0$ and/or $\text{scaleb}=0$:

- if $\text{scalea}=0$, there are no limitations for the values of $a[i]$ and $\text{shifta}$.
- if $\text{scaleb}=0$, there are no limitations for the values of $b[i]$ and $\text{shiftb}$.

Example

To use the $\text{v?LinearFrac}$ to shift vector $a$ by a scalar value, set $\text{scaleb}$ to 0. Note that even if $\text{scaleb}$ is 0, $b$ must be declared.

```c
#include <stdio.h>
#include "mkl_vml.h"

int main()
{
    double a[10], *b;
    double r[10];
    double scalea = 1.0, scaleb = 0.0;
    double shifta = -1.0, shiftb = 1.0;

    MKL_INT i=0,n=10;

    a[0]=-10000.0000;
    a[1]=-7777.7777;
    a[2]=-5555.5555;
    a[3]=-3333.3333;
    a[4]=-1111.1111;
    a[5]=1111.1111;
    a[6]=3333.3333;
    a[7]=5555.5555;
    a[8]=7777.7777;
    a[9]=10000.0000;

    vdLinearFrac( n, a, b, scalea, shifta, scaleb, shiftb, r );

    for(i=0;i<10;i++) {
        printf("%25.14f %25.14f\n",a[i],r[i]);
    }

    return 0;
}
```
To use the `v?LinearFrac` to compute \( \text{shifta}/(\text{scaleb} \cdot b[i] + \text{shiftb}) \), set \( \text{scalea} \) to 0. Note that even if \( \text{scalea} \) is 0, \( a \) must be declared.

**v?Fmod**

The `v?Fmod` function performs element by element computation of the modulus function of vector \( a \) with respect to vector \( b \).

**Syntax**

vsFmod \((n, a, b, y)\);

vmsFmod \((n, a, b, y, mode)\);

vdFmod \((n, a, b, y)\);

vmdFmod \((n, a, b, y, mode)\);

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a, b )</td>
<td>const float*</td>
<td>for vsFmod</td>
</tr>
<tr>
<td></td>
<td>const float*</td>
<td>for vmsFmod</td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td>for vdFmod</td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td>for vmdFmod</td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM ( mode ) setting for this function call. See <code>vmlSetMode</code> for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float*</td>
<td>for vsFmod</td>
</tr>
<tr>
<td></td>
<td>float*</td>
<td>for vmsFmod</td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td>for vdFmod</td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td>for vmdFmod</td>
</tr>
</tbody>
</table>

**Description**

The `v?Fmod` function computes the modulus function of each element of vector \( a \), with respect to the corresponding elements of vector \( b \):

\[
a_i - b_i \cdot \text{trunc}(a_i/b_i)
\]

In general, the modulus function \( \text{fmod} \ (a_i, b_i) \) returns the value \( a_i - n \cdot b_i \) for some integer \( n \) such that if \( b_i \) is nonzero, the result has the same sign as \( a_i \) and a magnitude less than the magnitude of \( b_i \).
Special values for Real Function v?Fmod(x, y)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>x not NAN</td>
<td>±0</td>
<td>NAN</td>
<td>VML_STATUS_SING</td>
<td>INVALID</td>
</tr>
<tr>
<td>±∞</td>
<td>y not NAN</td>
<td>±0</td>
<td>VML_STATUS_SING</td>
<td>INVALID</td>
</tr>
<tr>
<td>±0</td>
<td>y ≠ 0, not NAN</td>
<td>±0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x finite</td>
<td>±∞</td>
<td>x</td>
<td></td>
<td>UNDERFLOW if x is subnormal</td>
</tr>
<tr>
<td>NAN</td>
<td>y</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>NAN</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE**
If element $i$ in the result of v?Fmod is 0, its sign is that of $a_i$.

**See Also**
- **Div** Performs element by element division of vector $a$ by vector $b$
- **Remainder** Performs element by element computation of the remainder function on the elements of vector $a$ and the corresponding elements of vector $b$.

**v?Remainder**
*Performs element by element computation of the remainder function on the elements of vector $a$ and the corresponding elements of vector $b.*

**Syntax**
- `vsRemainder (n, a, b, y);`
- `vmsRemainder (n, a, b, y, mode);`
- `vdRemainder (n, a, b, y);`
- `vmdRemainder (n, a, b, y, mode);`

**Include Files**
- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$, $b$</td>
<td>const float* for vsRemainder</td>
<td>Pointers to arrays containing the input vectors $a$ and $b$.</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsRemainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdRemainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdRemainder</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsRemainder</td>
<td>Pointer to an array containing the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsRemainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdRemainder</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdRemainder</td>
<td></td>
</tr>
</tbody>
</table>

Description

Computes the remainder of each element of vector $a$, with respect to the corresponding elements of vector $b$: compute the values of $n$ such that

$$n = a_i - n \times b_i$$

where $n$ is the integer nearest to the exact value of $a_i/b_i$. If two integers are equally close to $a_i/b_i$, $n$ is the even one. If $n$ is zero, it has the same sign as $a_i$.

Special values for Real Function v?Remainder($x$, $y$)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$ not NAN</td>
<td>±0</td>
<td>NaN</td>
<td>VML_STATUS_DOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>±∞</td>
<td>$y$ not NAN</td>
<td>NaN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>±0</td>
<td>$y \neq 0$, not NAN</td>
<td>±0</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>$x$ finite</td>
<td>±∞</td>
<td>$x$</td>
<td></td>
<td>UNDERFLOW if $x$ is subnormal</td>
</tr>
<tr>
<td>NAN</td>
<td>$y$</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x$</td>
<td>NAN</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE

If element $i$ in the result of v?Remainder is 0, its sign is that of $a_i$.

See Also

Div  Performs element by element division of vector $a$ by vector $b$

Fmod  The v?Fmod function performs element by element computation of the modulus function of vector $a$ with respect to vector $b$.

Power and Root Functions

v?Inv

Performs element by element inversion of the vector.

Syntax

vsInv( n, a, y );
vmsInv( n, a, y, mode );
vdInv( n, a, y );
vmdInv( n, a, y, mode );
Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsInv, vmsInv</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdInv, vmdInv</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsInv, vmsInv</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdInv, vmdInv</td>
<td></td>
</tr>
</tbody>
</table>

Description
The v?Inv function performs element by element inversion of the vector.

Special Values for Real Function v?Inv(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

v?Div
Performs element by element division of vector a by vector b

Syntax

vsDiv( n, a, b, y );
vmsDiv( n, a, b, y, mode );
vdDiv( n, a, b, y );
vmdDiv( n, a, b, y, mode );
vcDiv( n, a, b, y );
vmcDiv( n, a, b, y, mode );
vzDiv( n, a, b, y );
vmdDiv( n, a, b, y, mode );
Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsDiv, vmsDiv&lt;br&gt;const double* for vdDiv, vmdDiv&lt;br&gt;const MKL_Complex8* for vcDiv, vmcDiv&lt;br&gt;const MKL_Complex16* for vzDiv, vmzDiv</td>
<td>Pointers to arrays that contain the input vectors a and b.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Precision Overflow Thresholds for Real \(v?\text{Div}\) Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>abs(a[i]) &lt; abs(b[i]) * FLT_MAX</td>
</tr>
<tr>
<td>double precision</td>
<td>abs(a[i]) &lt; abs(b[i]) * DBL_MAX</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsDiv, vmsDiv&lt;br&gt;double* for vdDiv, vmdDiv&lt;br&gt;MKL_Complex8* for vcDiv, vmcDiv&lt;br&gt;MKL_Complex16* for vzDiv, vmzDiv</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The \(v?\text{Div}\) function performs element by element division of vector \(a\) by vector \(b\).

Special values for Real Function \(v?\text{Div}(x)\)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>X &gt; +0</td>
<td>+0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>X &gt; +0</td>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>X &lt; +0</td>
<td>+0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>X &lt; +0</td>
<td>-0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>QNAN</td>
<td>VML_STATUS_SING</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>QNAN</td>
<td>VML_STATUS_SING</td>
<td></td>
</tr>
<tr>
<td>X &gt; +0</td>
<td>+∞</td>
<td>+0</td>
<td>VML_STATUS_SING</td>
<td></td>
</tr>
<tr>
<td>X &gt; +0</td>
<td>-∞</td>
<td>-0</td>
<td>VML_STATUS_SING</td>
<td></td>
</tr>
</tbody>
</table>
Specifications for special values of the complex functions are defined according to the following formula
\[
\text{Div}(x_1+i*y_1, x_2+i*y_2) = (x_1+i*y_1)*(x_2-i*y_2)/(x_2*x_2+y_2*y_2).
\]
Overflow in a complex function occurs when \(x_2+i*y_2\) is not zero, \(x_1, x_2, y_1, y_2\) are finite numbers, but the real or imaginary part of the exact result is so large that it does not fit the target precision. In that case, the function returns \(\infty\) in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to VML_STATUS_OVERFLOW.

\textbf{vSqrt}

*Computes a square root of vector elements.*

**Syntax**

\[
\begin{align*}
\text{vsSqrt}\left( n, a, y \right) ; \\
\text{vmsSqrt}\left( n, a, y, \text{mode} \right) ; \\
\text{vdSqrt}\left( n, a, y \right) ; \\
\text{vmdSqrt}\left( n, a, y, \text{mode} \right) ; \\
\text{vcSqrt}\left( n, a, y \right) ; \\
\text{vmcSqrt}\left( n, a, y, \text{mode} \right) ; \\
\text{vzSqrt}\left( n, a, y \right) ; \\
\text{vmzSqrt}\left( n, a, y, \text{mode} \right) ;
\end{align*}
\]

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsSqrt, vmsSqrt; const double* for vdSqrt, vmdSqrt; const MKL_Complex8* for vcSqrt, vcSqrt; const MKL_Complex16* for vzSqrt, vmzSqrt</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
<tr>
<td>(\text{mode})</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsSqrt, vmsSqrt</td>
<td>Pointer to an array that contains the output</td>
</tr>
<tr>
<td></td>
<td>double* for vdSqrt, vmdSqrt</td>
<td>vector y.</td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8* for vcSqrt, vmcSqrt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16* for vzSqrt, vmzSqrt</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Sqrt function computes a square root of vector elements.

Special Values for Real Function v?Sqrt(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>X &lt; +0</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
<td></td>
</tr>
</tbody>
</table>

See the Special Value Notations section for the conventions used in the table above.

Special Values for Complex Function v?Sqrt(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞</td>
<td>i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
</tr>
<tr>
<td>+iY</td>
<td>+0+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
<td>+∞+i∞</td>
</tr>
<tr>
<td>+i0</td>
<td>+0+i∞</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
</tr>
<tr>
<td>-i0</td>
<td>+0-i∞</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
</tr>
<tr>
<td>-iY</td>
<td>+0-i∞</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
</tr>
<tr>
<td>-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
<td>+∞-i∞</td>
</tr>
<tr>
<td>+iNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
</tr>
</tbody>
</table>

Notes:

- raises INVALID exception when the real or imaginary part of the argument is SNAN
- Sqrt(CONJ(z)) = CONJ(Sqrt(z)).

v?InvSqrt

Computes an inverse square root of vector elements.

Syntax

vsInvSqrt( n, a, y );
vmsInvSqrt( n, a, y, mode );
vdInvSqrt( n, a, y );

vmdInvSqrt( n, a, y, mode );

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>for vsInvSqrt,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmsInvSqrt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdInvSqrt,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmdInvSqrt</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float*</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>for vsInvSqrt,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmsInvSqrt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdInvSqrt,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmdInvSqrt</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?InvSqrt function computes an inverse square root of vector elements.

Special Values for Real Function v?InvSqrt(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>X &lt; +0</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

v?Cbrt

Computes a cube root of vector elements.

Syntax

vsCbrt( n, a, y );
vmsCbrt( n, a, y, mode );
vdcbrt( n, a, y );
vmdCbrt( n, a, y, mode );
Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsCbrt, vmsCbrt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdCbrt, vmdCbrt</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsCbrt, vmsCbrt</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdCbrt, vmdCbrt</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description
The v?Cbrt function computes a cube root of vector elements.

Special Values for Real Function v?Cbrt(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

v?InvCbrt
Computes an inverse cube root of vector elements.

Syntax
vsInvCbrt( n, a, y );
vmsInvCbrt( n, a, y, mode );
vdInvCbrt( n, a, y );
vmdInvCbrt( n, a, y, mode );

Include Files
• mkl.h
**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsInvCbrt, vmsInvCbrt; const double* for vdInvCbrt, vmdInvCbrt</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsInvCbrt, vmsInvCbrt; double* for vdInvCbrt, vmdInvCbrt</td>
<td>Pointer to an array that contains the output vector (y).</td>
</tr>
</tbody>
</table>

**Description**

The `v?InvCbrt` function computes an inverse cube root of vector elements.

**Special Values for Real Function v?InvCbrt(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+(\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-(\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+(\infty)</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-(\infty)</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**v?Pow2o3**

_Computes the cube root of the square of each vector element._

**Syntax**

```c
vsPow2o3( n, a, y );
vmSPow2o3( n, a, y, mode );
vDPow2o3( n, a, y );
vmdPow2o3( n, a, y, mode );
```

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsPow2o3,</td>
<td>Pointers to arrays that contain the input vector (a).</td>
</tr>
<tr>
<td></td>
<td>vmsPow2o3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdPow2o3,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmdPow2o3</td>
<td></td>
</tr>
<tr>
<td>(mode)</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See (vmlSetMode) for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsPow2o3, vmsPow2o3</td>
<td>Pointer to an array that contains the output vector (y).</td>
</tr>
<tr>
<td></td>
<td>double* for vdPow2o3, vmdPow2o3</td>
<td></td>
</tr>
</tbody>
</table>

Description

The \(v?Pow2o3\) function computes the cube root of the square of each vector element.

Special Values for Real Function \(v?Pow2o3(x)\)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+(\infty)</td>
<td>+(\infty)</td>
<td></td>
</tr>
<tr>
<td>-(\infty)</td>
<td>+(\infty)</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

\(v?Pow3o2\)

Computes the square root of the cube of each vector element.

Syntax

\[
\text{vsPow3o2}( n, a, y );
\]

\[
\text{vmsPow3o2}( n, a, y, mode );
\]

\[
\text{vdPow3o2}( n, a, y );
\]

\[
\text{vmdPow3o2}( n, a, y, mode );
\]

Include Files

- \(mkl.h\)
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsPow3o2, vmsPow3o2 const double* for vdPow3o2, vmdPow3o2</td>
<td>Pointers to arrays that contain the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Precision Overflow Thresholds for Pow3o2 Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>( \text{abs}(a[i]) &lt; \left( \text{FLT_MAX} \right)^{2/3} )</td>
</tr>
<tr>
<td>double precision</td>
<td>( \text{abs}(a[i]) &lt; \left( \text{DBL_MAX} \right)^{2/3} )</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsPow3o2, vmsPow3o2 double* for vdPow3o2, vmdPow3o2</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The \( v?\text{Pow3o2} \) function computes the square root of the cube of each vector element.

Special Values for Real Function \( v?\text{Pow3o2}(x) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X &lt; +0 )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(+0)</td>
<td>(+0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-0)</td>
<td>(-0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( -\infty )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>( +\infty )</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( v?\text{Pow} \)

*Computes a to the power b for elements of two vectors.*

Syntax

\[
\begin{align*}
\text{vsPow}( n, a, b, y ); \\
\text{vmsPow}( n, a, b, y, \text{mode} ); \\
\text{vdPow}( n, a, b, y ); \\
\text{vmdPow}( n, a, b, y, \text{mode} );
\end{align*}
\]
vcPow( n, a, b, y );
vmcPow( n, a, b, y, mode );
vzPow( n, a, b, y );
vmzPow( n, a, b, y, mode );

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsPow, vmsPow</td>
<td>Pointers to arrays that contain the input vectors a and b.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdPow, vmdPow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex8* for vcPow, vmcPow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16* for vzPow, vmzPow</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Precision Overflow Thresholds for Real v?Pow Function**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>( \text{abs}(a[i]) &lt; (\text{FLT}_\text{MAX})^{1/b[i]} )</td>
</tr>
<tr>
<td>double precision</td>
<td>( \text{abs}(a[i]) &lt; (\text{DBL}_\text{MAX})^{1/b[i]} )</td>
</tr>
</tbody>
</table>

Precision overflow thresholds for the complex v?Pow function are beyond the scope of this document.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsPow, vmsPow</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdPow, vmdPow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8* for vcPow, vmcPow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16* for vzPow, vmzPow</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The v?Pow function computes \( a \) to the power \( b \) for elements of two vectors.

The real function \( v(s/d)\text{Pow} \) has certain limitations on the input range of \( a \) and \( b \) parameters. Specifically, if \( a[i] \) is positive, then \( b[i] \) may be arbitrary. For negative \( a[i] \), the value of \( b[i] \) must be an integer (either positive or negative).

The complex function \( v(c/z)\text{Pow} \) has no input range limitations.
### Special values for Real Function vPow(x,y)

<table>
<thead>
<tr>
<th>Argument 1 (X)</th>
<th>Argument 2 (Y)</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>neg. odd integer</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>neg. odd integer</td>
<td>−∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>neg. even integer</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>neg. even integer</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>neg. non-integer</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>pos. odd integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>pos. odd integer</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>pos. even integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>pos. even integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>pos. non-integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>pos. non-integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>+∞</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>−∞</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>any value</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>+∞</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>−∞</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>QNAN</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>any value</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−∞</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>any value</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−∞</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>−0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X &lt; +0</td>
<td>non-integer</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>&lt; 1</td>
<td>−∞</td>
<td>+∞</td>
</tr>
<tr>
<td>+0</td>
<td>−∞</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>−∞</td>
<td>+∞</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>&gt; 1</td>
<td>−∞</td>
<td>+0</td>
</tr>
<tr>
<td>+∞</td>
<td>−∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>−∞</td>
<td>−∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>&lt; 1</td>
<td>+∞</td>
<td>+0</td>
</tr>
<tr>
<td>+0</td>
<td>+∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+∞</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>&gt; 1</td>
<td>+∞</td>
<td>+∞</td>
</tr>
<tr>
<td>Argument 1 (X)</td>
<td>Argument 2 (Y)</td>
<td>Result</td>
<td>VM Error Status</td>
<td>Exception</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------</td>
<td>--------</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>neg. odd integer</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>neg. even integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>neg. non-integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>pos. odd integer</td>
<td>-∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>pos. even integer</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>pos. non-integer</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>X &lt; +0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>X &gt; +0</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Big finite value*</td>
<td>Big finite value*</td>
<td>+/-∞</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

The complex double precision versions of this function, `vzPow` and `vmzPow`, are implemented in the EP accuracy mode only. If used in HA or LA mode, `vzPow` and `vmzPow` set the VM Error Status to `VML_STATUS_ACCURACYWARNING` (see the Values of the VM Status table).

* Overflow in a real function is supported only in the HA/LA accuracy modes. The overflow occurs when x and y are finite numbers, but the result is too large to fit the target precision. In this case, the function:

1. Returns $\infty$ in the result.
2. Raises the OVERFLOW exception.
3. Sets the VM Error Status to `VML_STATUS_OVERFLOW`.

Overflow in a complex function occurs (supported in the HA/LA accuracy modes only) when all $\text{RE}(x), \text{RE}(y), \text{IM}(x), \text{IM}(y)$ arguments are finite numbers, but the real or imaginary part of the computed result is so large that it does not fit the target precision. In this case, the function returns $\infty$ in that part of the result, raises the OVERFLOW exception, and sets the VM Error Status to `VML_STATUS_OVERFLOW` (overriding any possible `VML_STATUS_ACCURACYWARNING` status).

**vPowx**

*Computes vector a to the scalar power b.*

**Syntax**

```c
vsPowx( n, a, b, y );
vmsPowx( n, a, b, y, mode );
vdPowx( n, a, b, y );
vmdPowx( n, a, b, y, mode );
vcPowx( n, a, b, y );
vncPowx( n, a, b, y, mode );
vzPowx( n, a, b, y );
vmzPowx( n, a, b, y, mode );
```
Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsPowx, vmsPowx, const double* for vdPowx, vmdPowx, const MKL_Complex8* for vcPowx, vmcPowx, const MKL_Complex16* for vzPowx, vmzPowx</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>b</td>
<td>const float for vsPowx, vmsPowx, const double for vdPowx, vmdPowx, const MKL_Complex8 for vcPowx, vmcPowx, const MKL_Complex16 for vzPowx, vmzPowx</td>
<td>Constant value for power b.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Precision Overflow Thresholds for Real vPowx Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>( \abs{a[i]} &lt; \left( \text{FLT}_\text{MAX} \right)^{1/b} )</td>
</tr>
<tr>
<td>double precision</td>
<td>( \abs{a[i]} &lt; \left( \text{DBL}_\text{MAX} \right)^{1/b} )</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsPowx, vmsPowx, double* for vdPowx, vmdPowx, MKL_Complex8* for vcPowx, vmcPowx, MKL_Complex16* for vzPowx, vmzPowx</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The \( v\text{Powx} \) function computes \( a \) to the power \( b \) for a vector \( a \) and a scalar \( b \).

The real function \( v(s/d)\text{Powx} \) has certain limitations on the input range of \( a \) and \( b \) parameters. Specifically, if \( a[i] \) is positive, then \( b \) may be arbitrary. For negative \( a[i] \), the value of \( b \) must be an integer (either positive or negative).
The complex function $v(c/z)^{\text{Powx}}$ has no input range limitations.

Special values and VM Error Status treatment are the same as for the $v?\text{Pow}$ function.

$v?\text{Powr}$

*Computes* $a$ *to the power* $b$ *for elements of two vectors, where the elements of vector argument* $a$ *are all non-negative.*

**Syntax**

```c
vsPowr(n, a, b, y);
vmsPowr(n, a, b, y, mode);
vdPowr(n, a, b, y);
vmdPowr(n, a, b, y, mode);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a, b$</td>
<td>const float* for vsPowr</td>
<td>Pointers to arrays containing the input vectors $a$ and $b$.</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsPowr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdPowr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdPowr</td>
<td></td>
</tr>
<tr>
<td>$mode$</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsPowr</td>
<td>Pointer to an array containing the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsPowr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdPowr</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdPowr</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The $v?\text{Powr}$ function raises each element of vector $a$ by the corresponding element of vector $b$. The elements of $a$ are all nonnegative ($a_i \geq 0$).

**Precision Overflow Thresholds for Real Function $v?\text{Powr}$**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>$a_i &lt; (\text{FLT_MAX})^{1/b_i}$</td>
</tr>
</tbody>
</table>
### Data Type

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>double precision</td>
<td>( a_i &lt; (\text{DBL_MAX})^{1/b} )</td>
</tr>
</tbody>
</table>

Special values and VM Error Status treatment for \( v\text{Powr} \) function are the same as for \( v\text{Pow} \), unless otherwise indicated in this table:

### Special values for Real Function \( v\text{Powr}(x) \)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt; 0 )</td>
<td>any value ( y )</td>
<td>NAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>( 0 &lt; x &lt; \infty )</td>
<td>( \pm 0 )</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \pm 0 )</td>
<td>( -\infty &lt; y &lt; 0 )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \pm 0 )</td>
<td>( -\infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \pm 0 )</td>
<td>( y &gt; 0 )</td>
<td>( +0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 )</td>
<td>( -\infty &lt; y &lt; \infty )</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \pm 0 )</td>
<td>( \pm 0 )</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 1 )</td>
<td>( +\infty )</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x \geq 0 )</td>
<td>NAN</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{NAN} )</td>
<td>any value ( y )</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0 &lt; x &lt; 1 )</td>
<td>( -\infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x &gt; 1 )</td>
<td>( -\infty )</td>
<td>( +0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( 0 \leq x &lt; 1 )</td>
<td>( +\infty )</td>
<td>( +0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x &gt; 1 )</td>
<td>( +\infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( +\infty )</td>
<td>( x &lt; +0 )</td>
<td>( +0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( +\infty )</td>
<td>( x &gt; +0 )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

### See Also

**Pow** Computes \( a \) to the power \( b \) for elements of two vectors.

**Powx** Computes vector \( a \) to the scalar power \( b \).

**v?Hypot**

*Computes a square root of sum of two squared elements.*

### Syntax

\[
\begin{align*}
\text{vsHypot} & (\ n,\ a,\ b,\ y\ ); \\
\text{vmsHypot} & (\ n,\ a,\ b,\ y,\ \text{mode}\ ); \\
\text{vdHypot} & (\ n,\ a,\ b,\ y\ ); \\
\text{vmdHypot} & (\ n,\ a,\ b,\ y,\ \text{mode}\ );
\end{align*}
\]

### Include Files

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsHypot, vmsHypot</td>
<td>Pointers to arrays that contain the input vectors a and b.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdHypot, vmdHypot</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Precision Overflow Thresholds for Hypot Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>abs(a[i]) &lt; sqrt(FLT_MAX)</td>
</tr>
<tr>
<td></td>
<td>abs(b[i]) &lt; sqrt(FLT_MAX)</td>
</tr>
<tr>
<td>double precision</td>
<td>abs(a[i]) &lt; sqrt(DBL_MAX)</td>
</tr>
<tr>
<td></td>
<td>abs(b[i]) &lt; sqrt(DBL_MAX)</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsHypot, vmsHypot</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdHypot, vmdHypot</td>
<td></td>
</tr>
</tbody>
</table>

Description

The function v?Hypot computes a square root of sum of two squared elements.

Special values for Real Function v?Hypot(x)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>+\infty</td>
<td>any value</td>
<td>+\infty</td>
<td></td>
</tr>
<tr>
<td>any value</td>
<td>+\infty</td>
<td>+\infty</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>any value</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>any value</td>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>any value</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>any value</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

Exponential and Logarithmic Functions

v?Exp

*Computes an exponential of vector elements.*
Syntax

vsExp( n, a, y );
vmsExp( n, a, y, mode );
vdExp( n, a, y );
vmdExp( n, a, y, mode );
vcExp( n, a, y );
vmcExp( n, a, y, mode );
vzExp( n, a, y );
vmzExp( n, a, y, mode );

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsExp, vmsExp const double* for vdExp, vmdExp const MKL_Complex8* for vcExp, vmcExp const MKL_Complex16* for vzExp, vmzExp</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Precision Overflow Thresholds for Real v?Exp Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>a[i] &lt; Ln( FLT_MAX )</td>
</tr>
<tr>
<td>double precision</td>
<td>a[i] &lt; Ln( DBL_MAX )</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsExp, vmsExp double* for vdExp, vmdExp MKL_Complex8* for vcExp, vmcExp MKL_Complex16* for vzExp, vmzExp</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>
Description

The \( v?\text{Exp} \) function computes an exponential of vector elements.

### Special Values for Real Function \( v?\text{Exp}(x) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X &gt; \text{overflow} )</td>
<td>( +\infty )</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>( X &lt; \text{underflow} )</td>
<td>+0</td>
<td>VML_STATUS_UNDERFLOW</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>( +\infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( -\infty )</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
<td></td>
</tr>
</tbody>
</table>

See the [Special Value Notations](#) section for the conventions used in the table below.

### Special Values for Complex Function \( v?\text{Exp}(z) \)

<table>
<thead>
<tr>
<th>( \text{RE}(z) )</th>
<th>( -\infty )</th>
<th>( -X )</th>
<th>( -0 )</th>
<th>( +0 )</th>
<th>( +X )</th>
<th>( +\infty )</th>
<th>( \text{NAN} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( +i\infty )</td>
<td>( +0+i0 )</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>( +iY )</td>
<td>( +0\text{CIS}(Y) )</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>( +i0 )</td>
<td>( +0\text{CIS}(0) )</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>QNAN+i0</td>
</tr>
<tr>
<td>( -i0 )</td>
<td>( +0\text{CIS}(0) )</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>+1+i0</td>
<td>QNAN+i0</td>
</tr>
<tr>
<td>( -i\cdot Y )</td>
<td>( +0\text{CIS}(Y) )</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>INVALID</td>
<td>QNAN+i0</td>
</tr>
<tr>
<td>( -i\cdot \infty )</td>
<td>( +0+i0 )</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>( +i\text{NAN} )</td>
<td>( +0+i0 )</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
</tbody>
</table>

Notes:

- raises the INVALID exception when real or imaginary part of the argument is SNAN
- raises the INVALID exception on argument \( z=-\infty+i\cdot \text{QNAN} \)
- raises the OVERFLOW exception and sets the VM Error Status to \text{VML\_STATUS\_OVERFLOW} in the case of overflow, that is, when both \( \text{RE}(z) \) and \( \text{IM}(z) \) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.

\( v?\text{Exp}2 \)

Computes the base 2 exponential of vector elements.

### Syntax

\[
\begin{align*}
\text{vsExp2} & \ (n, a, y); \\
\text{vmsExp2} & \ (n, a, y, \text{mode}); \\
\text{vdExp2} & \ (n, a, y); \\
\text{vmdExp2} & \ (n, a, y, \text{mode});
\end{align*}
\]
Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsExp2, const float* for vmsExp2, const double* for vdExp2, const double* for vmdExp2</td>
<td>Pointer to the array containing the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsExp2, float* for vmsExp2, double* for vdExp2, double* for vmdExp2</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
</tbody>
</table>

Description

The v?Exp2 function computes the base 2 exponential of vector elements.

**Precision Overflow Thresholds for Real Function v?Exp2**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>( a_i &lt; \log_2(\text{FLT}_\text{MAX}) )</td>
</tr>
<tr>
<td>double precision</td>
<td>( a_i &lt; \log_2(\text{DBL}_\text{MAX}) )</td>
</tr>
</tbody>
</table>

See Special Value Notations for the conventions used in this table:

**Special values for Real Function v?Exp2(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x &gt; \text{overflow} )</td>
<td>( +\infty )</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>( x &lt; \text{underflow} )</td>
<td>0</td>
<td>VML_STATUS_UNDERFLOW</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>( +\infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( -\infty )</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See Also

Exp Computes an exponential of vector elements.
Exp10 Computes the base 10 exponential of vector elements.

v?Exp10

Computes the base 10 exponential of vector elements.

Syntax

vsExp10 (n, a, y);
vmsExp10 (n, a, y, mode);
vdExp10 (n, a, y);
vmdExp10 (n, a, y, mode);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to the array containing the input vector a.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>for vsExp10</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>for vmsExp10</td>
</tr>
<tr>
<td>a</td>
<td>const double*</td>
<td>for vdExp10</td>
</tr>
<tr>
<td>a</td>
<td>const double*</td>
<td>for vmdExp10</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float*</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
<tr>
<td>y</td>
<td>float*</td>
<td>for vsExp10</td>
</tr>
<tr>
<td>y</td>
<td>float*</td>
<td>for vmsExp10</td>
</tr>
<tr>
<td>y</td>
<td>double*</td>
<td>for vdExp10</td>
</tr>
<tr>
<td>y</td>
<td>double*</td>
<td>for vmdExp10</td>
</tr>
</tbody>
</table>

Description

The v?Exp10 function computes the base 10 exponential of vector elements.

Precision Overflow Thresholds for Real Function v?Exp10

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>(a_i &lt; \log_{10}(\text{FLT_MAX}))</td>
</tr>
<tr>
<td>double precision</td>
<td>(a_i &lt; \log_{10}(\text{DBL_MAX}))</td>
</tr>
</tbody>
</table>

See Special Value Notations for the conventions used in this table:
Special values for Real Function v?Pow(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x &gt;$ overflow</td>
<td>$+\infty$</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>$x &lt;$ underflow</td>
<td>+0</td>
<td>VML_STATUS_UNDERFLOW</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\infty$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See Also

- **Exp** Computes an exponential of vector elements.
- **Exp2** Computes the base 2 exponential of vector elements.

**v?Expm1**

*Computes an exponential of vector elements decreased by 1.*

**Syntax**

- **vsExpm1**($n$, $a$, $y$);
- **vmsExpm1**($n$, $a$, $y$, mode);
- **vdExpm1**($n$, $a$, $y$);
- **vmdExpm1**($n$, $a$, $y$, mode);

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$</td>
<td>const float* for vsExpm1, vmsExpm1, const double* for vdExpm1, vmdExpm1</td>
<td>Pointer to an array that contains the input vector $a$.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Precision Overflow Thresholds for Expm1 Function**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>$a[i] &lt; \text{Ln( FLT_MAX )}$</td>
</tr>
<tr>
<td>double precision</td>
<td>$a[i] &lt; \text{Ln( DBL_MAX )}$</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsExpm1, vmsExpm1</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdExpm1, vmdExpm1</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Expm1 function computes an exponential of vector elements decreased by 1.

Special Values for Real Function v?Expm1(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X &gt; overflow</td>
<td>+∞</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

v?Ln

Computes natural logarithm of vector elements.

Syntax

vsLn( n, a, y );
vmsLn( n, a, y, mode );
vdLn( n, a, y );
vmdLn( n, a, y, mode );
vcLn( n, a, y );
vmcLn( n, a, y, mode );
vzLn( n, a, y );
vmzLn( n, a, y, mode );

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsLn, vmsLn, const double* for vdLn, vmdLn, const MKL_Complex8* for vcLn, vmcLn</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
</tbody>
</table>
**Name** | **Type** | **Description**
---|---|---
const MKL_Complex16* | for vzLn, vmzLn | Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

**Output Parameters**

**Name** | **Type** | **Description**
---|---|---
y | float* for vsLn, vmsLn | Pointer to an array that contains the output vector y.
double* for vdLn, vmdLn
MKL_Complex8* for vcLn, vmcLn
MKL_Complex16* for vzLn, vmzLn

**Description**

The v?Ln function computes natural logarithm of vector elements.

**Special Values for Real Function v?Ln(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X &lt; +0</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the Special Value Notations section for the conventions used in the table below.

**Special Values for Complex Function v?Ln(z)**

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+i∞</td>
<td>+∞+ 1 - 3π/4</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/4</td>
<td>+∞+iQNaN</td>
<td></td>
</tr>
<tr>
<td>+iY</td>
<td>+∞+iπ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+∞+i0</td>
<td>QNAN+iQNaN INVALID</td>
</tr>
<tr>
<td>+i0</td>
<td>+∞+iπ</td>
<td>-∞+iπ ZERODIVIDE</td>
<td>-∞+i0 ZERODIVIDE</td>
<td>+∞+i0</td>
<td>QNAN+iQNaN INVALID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i0</td>
<td>+∞+iπ</td>
<td>-∞-iπ ZERODIVIDE</td>
<td>-∞-i0 ZERODIVIDE</td>
<td>+∞-i0 QNAN+iQNaN INVALID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RE(z)</td>
<td>IM(z)</td>
<td>-∞</td>
<td>-X</td>
<td>-0</td>
<td>+0</td>
<td>+X</td>
<td>+∞</td>
<td>NAN</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td>-i·Y</td>
<td>+∞ - i·π</td>
<td> </td>
<td> </td>
<td> </td>
<td> </td>
<td> </td>
<td> </td>
<td>+∞ - i·0</td>
</tr>
<tr>
<td>-i·∞</td>
<td>+∞ - i·3π/4</td>
<td>+∞ - i·π/2</td>
<td>+∞ - i·π/2</td>
<td>+∞ - i·π/2</td>
<td>+∞ - i·π/2</td>
<td>+∞ - i·π/4</td>
<td>+∞ - i·QNAN</td>
<td></td>
</tr>
<tr>
<td>+i·NAN</td>
<td>+∞ + i·QNAN</td>
<td>QNAN+i·QNAN INVALID</td>
<td>QNAN+i·QNAN INVALID</td>
<td>QNAN+i·QNAN INVALID</td>
<td>QNAN+i·QNAN INVALID</td>
<td>+∞ + i·QNAN</td>
<td>QNAN+i·QNAN INVALID</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- Raises INVALID exception when real or imaginary part of the argument is SNAN

**v2Log2**

*Computes the base 2 logarithm of vector elements.*

**Syntax**

```c
vsLog2 (n, a, y);
vmsLog2 (n, a, y, mode);
vdLog2 (n, a, y);
vmdLog2 (n, a, y, mode);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsLog2, const float* for vmsLog2, const double* for vdLog2, const double* for vmdLog2</td>
<td>Pointer to the array containing the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsLog2, float* for vmsLog2, double* for vdLog2</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
</tbody>
</table>
The v?Log2 function computes the base 2 logarithm of vector elements.

See Special Value Notations for the conventions used in this table:

**Special values for Real Function v?Log2(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(&lt; +0)</td>
<td>QNAN</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>(-\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>(-\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>(-\infty)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**See Also**

Ln  Computes natural logarithm of vector elements.

Log10 Computes the base 10 logarithm of vector elements.

**v?Log10**

_Computes the base 10 logarithm of vector elements._

**Syntax**

vsLog10( n, a, y );
vmsLog10( n, a, y, mode );
vdLog10( n, a, y );
vmdLog10( n, a, y, mode );
vcLog10( n, a, y );
vmcLog10( n, a, y, mode );
vsLog10( n, a, y );
vmzLog10( n, a, y, mode );

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsLog10, vmsLog10</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
</tbody>
</table>
The \texttt{v?Log10} function computes the base 10 logarithm of vector elements.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for \texttt{vsLog10}, \texttt{vmsLog10}</td>
<td>Pointer to an array that contains the output vector ( y ).</td>
</tr>
<tr>
<td></td>
<td>double* for \texttt{vdLog10}, \texttt{vmdLog10}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8* for \texttt{vcLog10}, \texttt{vmcLog10}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16* for \texttt{vzLog10}, \texttt{vmzLog10}</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The \texttt{v?Log10} function computes the base 10 logarithm of vector elements.

**Special Values for Real Function \texttt{v?Log10}(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( X &lt; +0 )</td>
<td>\texttt{QNaN}</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>(-\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>(-\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>(-\infty)</td>
<td>\texttt{QNaN}</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+( \infty )</td>
<td>( +\infty )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{QNaN}</td>
<td>\texttt{QNaN}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\texttt{SNAN}</td>
<td>\texttt{QNaN}</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the \texttt{Special Value Notations} section for the conventions used in the table below.

**Special Values for Complex Function \texttt{v?Log10}(z)**

<table>
<thead>
<tr>
<th>\texttt{RE}(z) \texttt{IM}(z)</th>
<th>(-\infty)</th>
<th>(-X)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+X)</th>
<th>(+\infty)</th>
<th>\texttt{NAN}</th>
</tr>
</thead>
<tbody>
<tr>
<td>+( \infty )</td>
<td>(+\infty + 2 \frac{\pi}{\ln(10)})</td>
<td>(+\infty + \frac{\pi}{\ln(10)})</td>
<td>(+\infty + \frac{\pi}{2 \ln(10)})</td>
<td>(+\infty + \frac{\pi}{2 \ln(10)})</td>
<td>(+\infty + \frac{\pi}{2 \ln(10)})</td>
<td>(+\infty + \frac{\pi}{2 \ln(10)})</td>
<td>(+\infty + \frac{\pi}{2 \ln(10)})</td>
</tr>
<tr>
<td>+( i \cdot Y )</td>
<td>(+\infty + \frac{\pi}{\ln(10)})</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+( \infty + I \cdot 0 )</td>
<td>\texttt{QNaN} + \texttt{QNaN} INVALID</td>
</tr>
</tbody>
</table>

See the \texttt{Special Value Notations} section for the conventions used in the table above.
### Vector Mathematical Functions

#### v?Log1p

*Computes a natural logarithm of vector elements that are increased by 1.*

**Syntax**

```c
vsLog1p( n, a, y );
vmsLog1p( n, a, y, mode );
vdLog1p( n, a, y );
vmdLog1p( n, a, y, mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
| a     | const float* for vsLog1p, vmsLog1p  
const double* for vdLog1p, vmdLog1p | Pointer to an array that contains the input vector a. |
| mode  | const MKL_INT64           | Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description. |

**Notes:**

- raises INVALID exception when real or imaginary part of the argument is SNAN
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsLog1p, vmsLog1p double* for vdLog1p, vmdLog1p</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The `v?Log1p` function computes a natural logarithm of vector elements that are increased by 1.

Special Values for Real Function `v?Log1p(x)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>X &lt; -1</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`v?Logb`

*Computes the exponents of the elements of input vector a.*

Syntax

```c
vsLogb (n, a, y);
vmsLogb (n, a, y, mode);
vdLogb (n, a, y);
vmdLogb (n, a, y, mode);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsLogb const float* for vmsLogb const double* for vdLogb const double* for vmdLogb</td>
<td>Pointer to the array containing the input vector a.</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
mode | const MKL_INT64 | Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Output Parameters

Name | Type | Description
--- | --- | ---
y | float* for vsLogb | Pointer to an array containing the output vector y.
float* for vmsLogb
double* for vdLogb
double* for vmdLogb

Description

The v?Logb function computes the exponents of the elements of the input vector a. For each element a_i of vector a, this is the integral part of log_2|a_i|. The returned value is exact and is independent of the current rounding direction mode.

See Special Value Notations for the conventions used in this table:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>-\infty</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-\infty</td>
<td>VML_STATUS_ERRDOM</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-\infty</td>
<td>+\infty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+\infty</td>
<td>+\infty</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

Trigonometric Functions

v?Cos

*Computes cosine of vector elements.*

Syntax

vsCos( n, a, y );
vmsCos( n, a, y, mode );
vdCos( n, a, y );
vmdCos( n, a, y, mode );
vcCos( n, a, y );
vmcCos( n, a, y, mode );
vzCos( n, a, y );
vmzCos( n, a, y, mode );
Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
| a    | const float* for vsCos, vmsCos  
  const double* for vdCos, vmdCos  
  const MKL_Complex8* for vcCos, vmcCos  
  const MKL_Complex16* for vzCos, vmzCos | Pointer to an array that contains the input vector a. |
| mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description. |

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| y    | float* for vsCos, vmsCos  
  double* for vdCos, vmdCos  
  MKL_Complex8* for vcCos, vmcCos  
  MKL_Complex16* for vzCos, vmzCos | Pointer to an array that contains the output vector y. |

Description

The v?Cos function computes cosine of vector elements.

Note that arguments abs(a[i]) ≤ 2^{13} and abs(a[i]) ≤ 2^{16} for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?Cos(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

\[ \cos(z) = \cosh(i*z) \].
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for
optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and
SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or
effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-
dependent optimizations in this product are intended for use with Intel microprocessors. Certain
optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to
the applicable product User and Reference Guides for more information regarding the specific instruction
sets covered by this notice.

Notice revision #20110804

v?Sin

*Computes sine of vector elements.*

**Syntax**

vsin( n, a, y );
vmSin( n, a, y, mode );
vdSin( n, a, y );
vmdSin( n, a, y, mode );
vcSin( n, a, y );
vmcSin( n, a, y, mode );
vzSin( n, a, y );
vmzSin( n, a, y, mode );

**Include Files**

* mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex8*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16*</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| $y$  | float* for vsSin, vmsSin  
     double* for vdSin, vmdSin  
     MKL_Complex8* for vcSin, vmcSin  
     MKL_Complex16* for vzSin, vmzSin | Pointer to an array that contains the output vector $y$. |

**Description**

The function computes sine of vector elements.

Note that arguments $\text{abs}(a[i]) \leq 2^{13}$ and $\text{abs}(a[i]) \leq 2^{16}$ for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

**Special Values for Real Function $v?\sin(x)$**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+0$</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

$$\text{Sin}(z) = -i*\text{Sinh}(i*z).$$

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

$v?\sin\cos$

*Computes sine and cosine of vector elements.*

**Syntax**

```c
vsSinCos( n, a, y, z );
vmsSinCos( n, a, y, z, mode );
vdSinCos( n, a, y, z );
vmdSinCos( n, a, y, z, mode );
```
Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to an array that contains the input vector a. For vsSinCos, vmsSinCos, double* for vdSinCos, vmdSinCos</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y, z</td>
<td>float*</td>
<td>Pointers to arrays that contain the output vectors y (for sine values) and z (for cosine values). For vsSinCos, vmsSinCos, double* for vdSinCos, vmdSinCos</td>
</tr>
</tbody>
</table>

Description

The function computes sine and cosine of vector elements.

Note that arguments abs(a[i]) ≤ 2^{13} and abs(a[i]) ≤ 2^{16} for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function v?SinCos(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result 1</th>
<th>Result 2</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>QNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

Sin(z) = -i*Sinh(i*z).

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-
Optimization Notice

dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

v?CIS
Computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

Syntax
vcCIS( n, a, y );
vmcCIS( n, a, y, mode );
vzCIS( n, a, y );
vmzCIS( n, a, y, mode );

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>MKL_Complex8*</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16*</td>
<td></td>
</tr>
</tbody>
</table>

Description
The v?CIS function computes complex exponent of real vector elements (cosine and sine of real vector elements combined to complex value).

See the Special Value Notations section for the conventions used in the table below.
Special Values for Complex Function v?CIS(x)

<table>
<thead>
<tr>
<th>x</th>
<th>CIS(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ ∞</td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>+ 0</td>
<td>+1+i 0</td>
</tr>
<tr>
<td>- 0</td>
<td>+1-i 0</td>
</tr>
<tr>
<td>- ∞</td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>NaN</td>
<td>QNAN+i QNAN</td>
</tr>
</tbody>
</table>

Notes:
- raises INVALID exception when the argument is SNAN
- raises INVALID exception and sets the VM Error Status to VML_STATUS_ERRDOM for x=+∞, x=-∞

v?Tan
Computes tangent of vector elements.

Syntax
vsTan( n, a, y );
vmsTan( n, a, y, mode );
vdTan( n, a, y );
vdmdTan( n, a, y, mode );
vcTan( n, a, y );
vmdTan( n, a, y, mode );
vzTan( n, a, y );
vmzTan( n, a, y, mode );

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsTan, vmsTan, const double* for vdTan, vmdTan, const MKL_Complex8* for vcTan, vmdTan, const MKL_Complex16* for vzTan, vmzTan</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
y | float* for `vsTan`, `vmsTan` | Pointer to an array that contains the output vector `y`. |
double* for `vdTan`, `vmdTan` |
MKL_Complex8* for `vcTan`, `vmcTan` |
MKL_Complex16* for `vzTan`, `vmzTan` |

Description
The `v?Tan` function computes tangent of vector elements.

Note that arguments \( \text{abs}(a[i]) \leq 2^{13} \) and \( \text{abs}(a[i]) \leq 2^{16} \) for single and double precisions respectively are called fast computational path. These are trigonometric function arguments for which VM provides the best possible performance. Avoid arguments that do not belong to the fast computational path in the VM High Accuracy (HA) and Low Accuracy (LA) functions. Alternatively, you can use VM Enhanced Performance (EP) functions that are fast on the entire function domain. However, these functions provide less accuracy.

Special Values for Real Function `v?Tan(x)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+( \infty )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-( \infty )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

\[ \text{Tan}(z) = -i \cdot \text{Tanh}(i \cdot z). \]

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

`v?Acos`

*Computes inverse cosine of vector elements.*
Syntax

vsAcos( n, a, y );
vmsAcos( n, a, y, mode );
vdAcos( n, a, y );
vmdAcos( n, a, y, mode );
vCAcos( n, a, y );
vmcAcos( n, a, y, mode );
vzAcos( n, a, y );
vmzAcos( n, a, y, mode );

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const int</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsAcos, vmsAcos const double* for vdAcos, vmdAcos const MKL_Complex8* for vcAcos, vmcAcos const MKL_Complex16* for vzAcos, vmzAcos</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAcos, vmsAcos double* for vdAcos, vmdAcos MKL_Complex8* for vcAcos, vmcAcos MKL_Complex16* for vzAcos, vmzAcos</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The v?Acos function computes inverse cosine of vector elements.

Special Values for Real Function v?Acos(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+π/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Special Values for Complex Function v?Acos(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+i∞</td>
<td></td>
<td>-X+i∞</td>
<td>-0+i∞</td>
<td>+0+i∞</td>
<td>+X+i∞</td>
<td>+∞</td>
<td>QNAN+i∞</td>
<td></td>
</tr>
<tr>
<td>+i-∞</td>
<td></td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+i0</td>
<td></td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i0</td>
<td></td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>+0+i∞</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i-∞</td>
<td></td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i∞</td>
<td></td>
<td>-X+i∞</td>
<td>-X+i∞</td>
<td>-X+i∞</td>
<td>-X+i∞</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+iNAN</td>
<td></td>
<td>QNAN+i∞</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- raises INVALID exception when real or imaginary part of the argument is SNAN
- \( \text{Acos} (\text{CONJ}(z)) = \text{CONJ}(\text{Acos}(z)) \).

### v?Asin

**Computes inverse sine of vector elements.**

**Syntax**

```c
vsAsin( n, a, y );
vmsAsin( n, a, y, mode );
vdAsin( n, a, y );
vmdAsin( n, a, y, mode );
vcAsin( n, a, y );
vmcAsin( n, a, y, mode );
vzAsin( n, a, y );
vmzAsin( n, a, y, mode );
```
Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsAsin, vmsAsin&lt;br&gt;const double* for vdAsin, vmdAsin&lt;br&gt;const MKL_Complex8* for vcAsin, vmcAsin&lt;br&gt;const MKL_Complex16* for vzAsin, vmzAsin</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAsin, vmsAsin&lt;br&gt;double* for vdAsin, vmdAsin&lt;br&gt;MKL_Complex8* for vcAsin, vmcAsin&lt;br&gt;MKL_Complex16* for vzAsin, vmzAsin</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The v?Asin function computes inverse sine of vector elements.

Special Values for Real Function v?Asin(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>π/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>-π/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>&gt; 1</td>
<td>QNAN</td>
</tr>
<tr>
<td>+∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula

\[ \text{Asin}(z) = -i \text{Asinh}(i*z). \]
v?Atan

Computes inverse tangent of vector elements.

Syntax

vsAtan( n, a, y );
vmsAtan( n, a, y, mode );
vdAtan( n, a, y );
vmAtan( n, a, y, mode );
vcAtan( n, a, y );
vmcAtan( n, a, y, mode );
vzAtan( n, a, y );
vmzAtan( n, a, y, mode );

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsAtan, vmsAtan, const double* for vdAtan, vmdAtan, const MKL_Complex8* for vcAtan, vmcAtan, const MKL_Complex16* for vzAtan, vmzAtan</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAtan, vmsAtan, double* for vdAtan, vmdAtan, MKL_Complex8* for vcAtan, vmcAtan, MKL_Complex16* for vzAtan, vmzAtan</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The v?Atan function computes inverse tangent of vector elements.
**Special Values for Real Function v?Atan(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+π/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-π/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

Specifications for special values of the complex functions are defined according to the following formula:

\[ \text{Atan}(z) = -i \cdot \text{Atanh}(i \cdot z) \]

**v?Atan2**

*Computes four-quadrant inverse tangent of elements of two vectors.*

**Syntax**

vsAtan2( n, a, b, y );

vmsAtan2( n, a, b, y, mode );

vdAtan2( n, a, b, y );

vmdAtan2( n, a, b, y, mode );

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsAtan2, vmsAtan2,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdAtan2, vmdAtan2</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAtan2, vmsAtan2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdAtan2, vmdAtan2</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The v?Atan2 function computes four-quadrant inverse tangent of elements of two vectors.
The elements of the output vector $y$ are computed as the four-quadrant arctangent of $a[i] / b[i]$.

### Special values for Real Function v?Atan2(x)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-\infty$</td>
<td>$-\infty$</td>
<td>$-3\pi/4$</td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$X &lt; +0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$-0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$+0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$X &gt; +0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$+\infty$</td>
<td>$-\pi/4$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; +0$</td>
<td>$-\infty$</td>
<td>$-\pi$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; +0$</td>
<td>$-0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; +0$</td>
<td>$+0$</td>
<td>$-\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$X &lt; +0$</td>
<td>$+\infty$</td>
<td>$-0$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$-\infty$</td>
<td>$-\pi$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$X &lt; +0$</td>
<td>$-\pi$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$-0$</td>
<td>$-\pi$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$+0$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$X &gt; +0$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$+\infty$</td>
<td>$0$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$-\infty$</td>
<td>$+\pi$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$X &lt; +0$</td>
<td>$+\pi$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$-0$</td>
<td>$+\pi$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$+0$</td>
<td>$+0$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$X &gt; +0$</td>
<td>$+0$</td>
<td></td>
</tr>
<tr>
<td>$+0$</td>
<td>$+\infty$</td>
<td>$+0$</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>$-\infty$</td>
<td>$+\pi$</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>$-0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>$+0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>$+\infty$</td>
<td>$+0$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$-\infty$</td>
<td>$+3\pi/4$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$X &lt; +0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$-0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$X &gt; +0$</td>
<td>$+\pi/2$</td>
<td></td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\infty$</td>
<td>$+\pi/4$</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>$X &gt; +0$</td>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>$X &gt; +0$</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>$X &gt; +0$</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>
**v?Cospi**  
*Computes the cosine of vector elements multiplied by \( n \).*

**Syntax**

vsCospi \((n, a, y)\);

vmsCospi \((n, a, y, mode)\);

vdCospi \((n, a, y)\);

vmdCospi \((n, a, y, mode)\);

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a )</td>
<td>const float* for vsCospi</td>
<td>Pointer to the array containing the input vector ( a ).</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsCospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdCospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdCospi</td>
<td></td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float* for vsCospi</td>
<td>Pointer to an array containing the output vector ( y ).</td>
</tr>
<tr>
<td></td>
<td>float* for vmsCospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdCospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdCospi</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The \( v?\)Cospi function computes the cosine of vector elements multiplied by \( n \). For an argument \( x \), the function computes \( \cos(nx) \).

**Special values for Real Function \( v?\)Cospi(\( x \))**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( n + 0.5 ), for any integer ( n ) where ( n + 0.5 ) is representable</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Result</td>
<td>VM Error Status</td>
<td>Exception</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>-----------------</td>
<td>-----------</td>
</tr>
<tr>
<td>±∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**Application Notes**

If arguments \(|a_i| \leq 2^{22}\) for single precision or \(|a_i| \leq 2^{43}\) for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

**See Also**

**Cos** Computes cosine of vector elements.

**Cosd** Computes the cosine of vector elements multiplied by \(\pi/180\).

**v?Sinpi**

*Computes the sine of vector elements multiplied by \(n\).*

**Syntax**

```c
vsSinpi (n, a, y);
vmsSinpi (n, a, y, mode);
vdSinpi (n, a, y);
vmdSinpi (n, a, y, mode);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsSinpi</td>
<td>Pointer to the array containing the input vector (a).</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsSinpi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdSinpi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdSinpi</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsSinpi</td>
<td>Pointer to an array containing the output vector (y).</td>
</tr>
<tr>
<td></td>
<td>float* for vmsSinpi</td>
<td></td>
</tr>
</tbody>
</table>
### Description

The vSinpi function computes the sine of vector elements multiplied by \( n \). For an argument \( x \), the function computes \( \sin(n \cdot x) \).

### Special values for Real Function vSinpi(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+n, positive integer</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-n, negative integer</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>±∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Application Notes

If arguments \( \text{abs}(a_i) \leq 2^{23} \) for single precision or \( \text{abs}(a_i) \leq 2^{51} \) for double precision, they belong to the fast computational path: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

### See Also

- **Sin** Computes sine of vector elements.
- **Sind** Computes the sine of vector elements multiplied by \( n/180 \).

### vTanpi

*Computes the tangent of vector elements multiplied by \( n \).*

### Syntax

```c
vsTanpi (n, a, y);
vmsTanpi (n, a, y, mode);
vdTanpi (n, a, y);
vmdTanpi (n, a, y, mode);
```

### Include Files

- mkl.h

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
**Name** | **Type** | **Description**
---|---|---
\(a\) | const float* for vsTanpi, const float* for vmsTanpi, const double* for vdTanpi, const double* for vmdTanpi | Pointer to the array containing the input vector \(a\).
\(mode\) | const MKL_INT64 | Overrides the global VM \(mode\) setting for this function call. See \(vmlSetMode\) for possible values and their description.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
\(y\) | float* for vsTanpi, float* for vmsTanpi, double* for vdTanpi, double* for vmdTanpi | Pointer to an array containing the output vector \(y\).

**Description**

The \(v?\text{Tanpi}\) function computes the tangent of vector elements multiplied by \(n\). For an argument \(x\), the function computes \(\tan(\pi \cdot x)\).

**Special values for Real Function \(v?\text{Tanpi}(x)\)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
</table>
\(+0\) | +1 | | |
\(-0\) | +1 | | |
\(\pm\infty\) | QNAN | VML_STATUS_ERRDOM | INVALID |
\(n\), even integer | \(\text{copysign}(0.0, n)\) | | |
\(n\), odd integer | \(\text{copysign}(0.0, -n)\) | | |
\(n + 0.5\), for \(n\) even integer and \(n + 0.5\) representable | \(+\infty\) | | |
\(n + 0.5\), for \(n\) odd integer and \(n + 0.5\) representable | \(-\infty\) | | |
\(\text{QNAN}\) | \(\text{QNAN}\) | | |
\(\text{SNAN}\) | \(\text{QNAN}\) | | INVALID |

The \(\text{copysign}(x, y)\) function returns the first vector argument \(x\) with the sign changed to match that of the second argument \(y\).

**Application Notes**

If arguments \(\text{abs}(a_i) \leq 2^{13}\) for single precision or \(\text{abs}(a_i) \leq 2^{67}\) for double precision, they belong to the fast computational path: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

**See Also**

\(\text{Tan}\) Computes tangent of vector elements.
**Tand** Computes the tangent of vector elements multiplied by \( n/180 \).

**v?Acospi**

*Computes the inverse cosine of vector elements divided by \( n \).*

**Syntax**

\[
\begin{align*}
\text{vsAcospi} & \ (n, \ a, \ y); \\
\text{vmsAcospi} & \ (n, \ a, \ y, \ mode); \\
\text{vdAcospi} & \ (n, \ a, \ y); \\
\text{vmdAcospi} & \ (n, \ a, \ y, \ mode);
\end{align*}
\]

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a )</td>
<td>const float* for vsAcospi</td>
<td>Pointer to the array containing the input vector ( a ).</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsAcospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdAcospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdAcospi</td>
<td></td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM ( mode ) setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float* for vsAcospi</td>
<td>Pointer to an array containing the output vector ( y ).</td>
</tr>
<tr>
<td></td>
<td>float* for vmsAcospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdAcospi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdAcospi</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The **v?Acospi** function computes the inverse cosine of vector elements divided by \( n \). For an argument \( x \), the function computes \( \text{acos}(x)/n \).

See **Special Value Notations** for the conventions used in this table:

**Special values for Real Function v?Acospi(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Argument | Result | VM Error Status | Exception
---|---|---|---
+1 | +0 | | |
-1 | +1 | | |
|x| > 1 | QNAN | VML_STATUS_ERRDOM | INVALID |
+∞ | QNAN | VML_STATUS_ERRDOM | INVALID |
-∞ | QNAN | VML_STATUS_ERRDOM | INVALID |
QNAN | QNAN | | |
SNAN | QNAN | | |

#### See Also

**Acos**  Computes inverse cosine of vector elements.

**v?Asinpi**

*Computes the inverse sine of vector elements divided by n.*

#### Syntax

vsAsinpi \((n, a, y)\);

vmsAsinpi \((n, a, y, mode)\);

vdAsinpi \((n, a, y)\);

vmdAsinpi \((n, a, y, mode)\);

#### Include Files

- mkl.h

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsAsinpi, const float* for vmsAsinpi, const double* for vdAsinpi, const double* for vmdAsinpi</td>
<td>Pointer to the array containing the input vector (a).</td>
</tr>
<tr>
<td>(mode)</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See <strong>vmlSetMode</strong> for possible values and their description.</td>
</tr>
</tbody>
</table>

#### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsAsinpi, float* for vmsAsinpi, double* for vdAsinpi</td>
<td>Pointer to an array containing the output vector (y).</td>
</tr>
</tbody>
</table>
### Description

The `v?Asinpi` function computes the inverse sine of vector elements divided by \( n \). For an argument \( x \), the function computes \( \text{asin}(x)/n \).

### Special values for Real Function `v?Asinpi(x)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>+1/2</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-1</td>
<td>-1/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### See Also

- **Asin**: Computes inverse sine of vector elements.

- **v?Atanpi**: Computes the inverse tangent of vector elements divided by \( n \).

### Syntax

- `vsAtanpi (n, a, y);`
- `vmsAtanpi (n, a, y, mode);`
- `vdAtanpi (n, a, y);`
- `vmdAtanpi (n, a, y, mode);`

### Include Files

- `mkl.h`

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td><code>const MKL_INT</code></td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td><code>a</code></td>
<td><code>const float* for vsAtanpi</code></td>
<td>Pointer to the array containing the input vector <code>a</code>.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAtanpi</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsAtanpi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdAtanpi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdAtanpi</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The v?Atanpi function computes the inverse tangent of vector elements divided by \( n \). For an argument \( x \), the function computes \( \text{atan}(x)/n \).

**Special values for Real Function v?Atanpi(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+( \infty )</td>
<td>+1/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>-1/2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

Atan    Computes inverse tangent of vector elements.

v?Atan2pi

Computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by \( n \).

**Syntax**

vsAtan2pi \( (n, a, b, y) \);

vmsAtan2pi \( (n, a, b, y, mode) \);

vdAtan2pi \( (n, a, b, y) \);

vmdAtan2pi \( (n, a, b, y, mode) \);

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsAtan2pi, const float* for vmsAtan2pi, const double* for vdAtan2pi, const double* for vmdAtan2pi</td>
<td>Pointers to the arrays containing the input vectors a and b.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAtan2pi, float* for vmsAtan2pi, double* for vdAtan2pi, double* for vmdAtan2pi</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
</tbody>
</table>

Description

The v?Atan2pi function computes the four-quadrant inverse tangent of the ratios of the corresponding elements of two vectors divided by n.

For the elements of the output vector y, the function computes the four-quadrant arctangent of \( a_i / b_i \), with the result divided by \( n \).

Special values for Real Function v?Atan2pi(x, y)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-( \infty )</td>
<td>-( \infty )</td>
<td>-3/4</td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>x &lt; +0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>0</td>
<td>+1/2</td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>+0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>x &gt; +0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>-( \infty )</td>
<td>+( \infty )</td>
<td>-1/4</td>
<td></td>
</tr>
<tr>
<td>x &lt; +0</td>
<td>-( \infty )</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>x &lt; +0</td>
<td>-0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>x &lt; +0</td>
<td>+0</td>
<td>-1/2</td>
<td></td>
</tr>
<tr>
<td>x &lt; +0</td>
<td>+( \infty )</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-( \infty )</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>x &lt; +0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>x &gt; +0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+( \infty )</td>
<td>-0</td>
<td></td>
</tr>
</tbody>
</table>
**See Also**

**Atan2** Computes four-quadrant inverse tangent of elements of two vectors.

**v?Cosd**

*Computes the cosine of vector elements multiplied by \( \pi/180 \).*

**Syntax**

- `vsCosd (n, a, y);`
- `vmsCosd (n, a, y, mode);`
- `vdCosd (n, a, y);`
- `vmdCosd (n, a, y, mode);`

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td><code>const MKL_INT</code></td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
**Name** | **Type** | **Description**
--- | --- | ---
\( a \) | `const float*` for vsCosd | Pointer to the array containing the input vector \( a \).
| `const float*` for vmsCosd | |
| `const double*` for vdCosd | |
| `const double*` for vmdCosd | |

\( mode \) | `const MKL_INT64` | Overrides the global VM \( mode \) setting for this function call. See \texttt{vmlSetMode} for possible values and their description.

**Output Parameters**

**Name** | **Type** | **Description**
--- | --- | ---
y | `float*` for vsCosd | Pointer to an array containing the output vector \( y \).
| `float*` for vmsCosd | |
| `double*` for vdCosd | |
| `double*` for vmdCosd | |

**Description**

The \( v?\text{Cosd} \) function computes the cosine of vector elements multiplied by \( n/180 \). For an argument \( x \), the function computes \( \cos(\pi \times x / 180) \).

**Special values for Real Function \( v?\text{Cosd}(x) \)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \pm \infty )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**Application Notes**

If arguments \( \text{abs}(a_i) \leq 2^{24} \) for single precision or \( \text{abs}(a_i) \leq 2^{52} \) for double precision, they belong to the **fast computational path**: arguments for which VM provides the best possible performance. Avoid arguments with do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

**See Also**

\texttt{Cos} Computes cosine of vector elements.

\texttt{Cospi} Computes the cosine of vector elements multiplied by \( n \).

\texttt{v?Sind} Computations of sine of vector elements.

**Syntax**

\texttt{vsSind} \((n, a, y)\);

\texttt{vmsSind} \((n, a, y, mode)\);
vdSind \((n, a, y)\);

vmdSind \((n, a, y, \text{mode})\);

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsSind, const double* for vmsSind, const double* for vmdSind</td>
<td>Pointer to the array containing the input vector (a).</td>
</tr>
<tr>
<td>(mode)</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See \texttt{vmlSetMode} for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsSind, float* for vmsSind, double* for vmdSind</td>
<td>Pointer to an array containing the output vector (y).</td>
</tr>
</tbody>
</table>

**Description**

The \(v?\text{Sind}\) function computes the sine of vector elements multiplied by \(n/180\). For an argument \(x\), the function computes \(\sin(\pi * x / 180)\).

**Special values for Real Function \(v?\text{Sind}(x)\)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+0)</td>
<td>(+0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(-0)</td>
<td>(-0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\pm\infty)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**Application Notes**

If arguments \(\text{abs}(a_i) \leq 2^{24}\) for single precision or \(\text{abs}(a_i) \leq 2^{52}\) for double precision, they belong to the fast computational path: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.
See Also
Sin  Computes sine of vector elements.
Sinpi Computes the sine of vector elements multiplied by $n$.

v?Tand
Computes the tangent of vector elements multiplied by n/180.

Syntax

vsTand ($n$, $a$, $y$);
vmsTand ($n$, $a$, $y$, $mode$);
vdTand ($n$, $a$, $y$);
vmdTand ($n$, $a$, $y$, $mode$);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$</td>
<td>const float* for vsTand</td>
<td>Pointer to the array containing the input vector $a$.</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsTand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdTand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdTand</td>
<td></td>
</tr>
<tr>
<td>$mode$</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM $mode$ setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsTand</td>
<td>Pointer to an array containing the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsTand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdTand</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdTand</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Tand function computes the tangent of vector elements multiplied by $n/180$. For an argument $x$, the function computes $\tan(n \times x/180)$.

Special values for Real Function v?Tand($x$)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Result</td>
<td>VM Error Status</td>
<td>Exception</td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td>--------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>±∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

The copysign($x$, $y$) function returns the first vector argument $x$ with the sign changed to match that of the second argument $y$.

**Application Notes**

If arguments $\text{abs}(a_i) \leq 2^{38}$ for single precision or $\text{abs}(a_i) \leq 2^{67}$ for double precision, they belong to the *fast computational path*: arguments for which VM provides the best possible performance. Avoid arguments which do not belong to the fast computational path in VM High Accuracy (HA) or Low Accuracy (LA) functions. For arguments which do not belong to the fast computational path you can use VM Enhanced Performance (EP) functions, which are fast on the entire function domain. However, these functions provide lower accuracy.

**See Also**

*Tan* Computes tangent of vector elements.

*Tanpi* Computes the tangent of vector elements multiplied by $n$.

**Hyperbolic Functions**

*v?Cosh*

*Computes hyperbolic cosine of vector elements.*

**Syntax**

vsCosh( $n$, $a$, $y$ );

vmsCosh( $n$, $a$, $y$, mode );

vdCosh( $n$, $a$, $y$ );

vmdCosh( $n$, $a$, $y$, mode );

vcCosh( $n$, $a$, $y$ );

vmcCosh( $n$, $a$, $y$, mode );

vzCosh( $n$, $a$, $y$ );

vmzCosh( $n$, $a$, $y$, mode );

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$</td>
<td>const float* for vsCosh, vmsCosh const double* for vdCosh, vmdCosh</td>
<td>Pointer to an array that contains the input vector $a$.</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
const MKL_Complex8* for vcCosh, vmcCosh | | 
const MKL_Complex16* for vzCosh, vmzCosh | | 
mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.

Precision Overflow Thresholds for Real v?Cosh Function

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>-$\ln(\text{FLT_MAX})$-$\ln2$ &lt; $a[i]$ &lt; $\ln(\text{FLT_MAX})$+$\ln2$</td>
</tr>
<tr>
<td>double precision</td>
<td>-$\ln(\text{DBL_MAX})$-$\ln2$ &lt; $a[i]$ &lt; $\ln(\text{DBL_MAX})$+$\ln2$</td>
</tr>
</tbody>
</table>

Precision overflow thresholds for the complex v?Cosh function are beyond the scope of this document.

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
y | float* for vsCosh, vmsCosh | Pointer to an array that contains the output vector $y$. |
double* for vdCosh, vmdCosh | | 
MKL_Complex8* for vcCosh, vmcCosh | | 
MKL_Complex16* for vzCosh, vmzCosh | | 

Description

The v?Cosh function computes hyperbolic cosine of vector elements.

Special Values for Real Function v?Cosh(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$0</td>
<td>$+1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-0$</td>
<td>$+1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X &gt;$ overflow</td>
<td>$+\infty$</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>$X &lt;$ -overflow</td>
<td>$+\infty$</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>$+\infty$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>$+\infty$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the Special Value Notations section for the conventions used in the table below.

Special Values for Complex Function v?Cosh(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>$-\infty$</th>
<th>$-X$</th>
<th>$-0$</th>
<th>$+0$</th>
<th>$+X$</th>
<th>$+\infty$</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+i\infty$</td>
<td>$-i\infty$</td>
<td>QNAN+iQNAN</td>
<td>QNAN+i0</td>
<td>QNAN+i0</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQAN</td>
</tr>
</tbody>
</table>
\[ \text{RE}(z) + i \text{IM}(z) \]

<table>
<thead>
<tr>
<th>(+\infty)</th>
<th>(-\infty)</th>
<th>(-X)</th>
<th>(-0)</th>
<th>(+0)</th>
<th>(+X)</th>
<th>(+\infty)</th>
<th>(\text{NAN})</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+iY)</td>
<td></td>
<td>(+X+i0)</td>
<td>(+1+i0)</td>
<td>(+1+i0)</td>
<td>(+X+i0)</td>
<td>(+\infty+i0)</td>
<td>(\text{QNAN+iQNAN})</td>
</tr>
<tr>
<td>(+i0)</td>
<td>(+X+i0)</td>
<td>(+1+i0)</td>
<td>(+1+i0)</td>
<td>(+X+i0)</td>
<td>(+\infty+i0)</td>
<td>(\text{QNAN+i0})</td>
<td></td>
</tr>
<tr>
<td>(-i0)</td>
<td>(+X+i0)</td>
<td>(+1+i0)</td>
<td>(+1+i0)</td>
<td>(+X+i0)</td>
<td>(+\infty+i0)</td>
<td>(\text{QNAN+i0})</td>
<td></td>
</tr>
<tr>
<td>(-i\infty)</td>
<td>(+X+i\text{QNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+i0})</td>
<td>(\text{QNAN+i0})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td></td>
</tr>
<tr>
<td>(+i\text{NAN})</td>
<td>(+X+i\text{QNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td>(\text{QNAN+iQNAN})</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- raises the INVALID exception when the real or imaginary part of the argument is SNAN.
- raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW in the case of overflow, that is, when \(\text{RE}(z), \text{IM}(z)\) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- \(\text{Cosh} (\text{CONJ}(z)) = \text{CONJ} (\text{Cosh}(z))\)
- \(\text{Cosh}(-z) = \text{Cosh}(z)\).

**v?Sinh**

*Computes hyperbolic sine of vector elements.*

**Syntax**

```c
vsSinh( n, a, y );
vmsSinh( n, a, y, mode );
vdSinh( n, a, y );
vmdSinh( n, a, y, mode );
vcSinh( n, a, y );
vmcSinh( n, a, y, mode );
vzSinh( n, a, y );
vmzSinh( n, a, y, mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsSinh, vmsSinh const double* for vdSinh, vmdSinh</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>const MKL_Complex8*</td>
<td>for vcSinh, vmcSinh</td>
<td></td>
</tr>
<tr>
<td>const MKL_Complex16*</td>
<td>for vzSinh, vmzSinh</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See <code>vmlSetMode</code> for possible values and their description.</td>
</tr>
</tbody>
</table>

**Precision Overflow Thresholds for Real vSinh Function**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Threshold Limitations on Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>single precision</td>
<td>(-\ln(\text{FLT_MAX})-\ln2 &lt; a[i] &lt; \ln(\text{FLT_MAX})+\ln2)</td>
</tr>
<tr>
<td>double precision</td>
<td>(-\ln(\text{DBL_MAX})-\ln2 &lt; a[i] &lt; \ln(\text{DBL_MAX})+\ln2)</td>
</tr>
</tbody>
</table>

Precision overflow thresholds for the complex `vSinh` function are beyond the scope of this document.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float*</td>
<td>for <code>vsSinh</code>, <code>vmsSinh</code></td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td>for <code>vdSinh</code>, <code>vmdSinh</code></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8*</td>
<td>for <code>vcSinh</code>, <code>vcSinh</code></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16*</td>
<td>for <code>vzSinh</code>, <code>vmzSinh</code></td>
</tr>
</tbody>
</table>

**Description**

The `vSinh` function computes hyperbolic sine of vector elements.

**Special Values for Real Function vSinh(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X &gt; overflow</td>
<td>+∞</td>
<td><code>VML\_STATUS\_OVERFLOW</code></td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>X &lt; -overflow</td>
<td>-∞</td>
<td><code>VML\_STATUS\_OVERFLOW</code></td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the `Special Value Notations` section for the conventions used in the table below.

**Special Values for Complex Function vSinh(z)**

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+\infty)</td>
<td>-i∞</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>-0+i QNAN</td>
<td>+0+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>+∞+i QNAN</td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>−i∞</td>
<td>QNAN+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>-0+i QNAN</td>
<td>+0+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>+∞+i QNAN</td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td>RE(z)</td>
<td>-∞</td>
<td>-X</td>
<td>-0</td>
<td>+0</td>
<td>+X</td>
<td>+∞</td>
<td>NAN</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td></td>
</tr>
<tr>
<td>+iY</td>
<td>-∞+i0</td>
<td></td>
<td>-0+i0</td>
<td>+0+i0</td>
<td></td>
<td></td>
<td>+∞+i0</td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td>+i0</td>
<td>-∞+i0</td>
<td>-∞+i0</td>
<td>+0+i0</td>
<td></td>
<td></td>
<td></td>
<td>QNAN+i 0</td>
<td></td>
</tr>
<tr>
<td>-i0</td>
<td>-∞-i0</td>
<td>-0-i0</td>
<td>+0-i0</td>
<td></td>
<td></td>
<td></td>
<td>QNAN-i 0</td>
<td></td>
</tr>
<tr>
<td>-iY</td>
<td>-∞-iY</td>
<td>-∞+i0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>QNAN+i QNAN</td>
</tr>
<tr>
<td>-i∞</td>
<td>-∞+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>-0+i QNAN</td>
<td>+0+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>+∞+i QNAN</td>
<td>QNAN+i QNAN</td>
<td></td>
</tr>
<tr>
<td>+i-NAN</td>
<td>-∞+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>-0+i QNAN</td>
<td>+0+i QNAN</td>
<td>QNAN+i QNAN</td>
<td>+∞+i QNAN</td>
<td>QNAN+i QNAN</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

- raises the INVALID exception when the real or imaginary part of the argument is SNAN
- raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW in the case of overflow, that is, when RE(z), IM(z) are finite non-zero numbers, but the real or imaginary part of the exact result is so large that it does not meet the target precision.
- \( \text{Sinh(CONJ}(z)) = \text{CONJ} (\text{Sinh}(z)) \)
- \( \text{Sinh}(-z) = -\text{Sinh}(z) \).

v?Tanh

Computes hyperbolic tangent of vector elements.

Syntax

```c
vsTanh(n, a, y);
vmssTanh(n, a, y, mode);
vdTanh(n, a, y);
vmdTanh(n, a, y, mode);
vcTanh(n, a, y);
vmcTanh(n, a, y, mode);
vzTanh(n, a, y);
vmzTanh(n, a, y, mode);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsTanh, vmssTanh, const double* for vdTanh, vmdTanh</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
</tbody>
</table>
Name | Type | Description
---|---|---
const MKL_Complex8* for vcTanh, vmcTanh
const MKL_Complex16* for vzTanh, vmzTanh
mode | const MKL_INT64 | Overrides global VM mode setting for this function call. See `vmlSetMode` for possible values and their description.

**Output Parameters**

Name | Type | Description
---|---|---
y | float* for vsTanh, vmsTanh
double* for vdTanh, vmdTanh
MKL_Complex8* for vcTanh, vmcTanh
MKL_Complex16* for vzTanh, vmzTanh

**Description**

The `v?Tanh` function computes hyperbolic tangent of vector elements.

**Special Values for Real Function v?Tanh(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the `Special Value Notations` section for the conventions used in the table below.

**Special Values for Complex Function v?Tanh(z)**

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+i∞</td>
<td>-1+i0</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN</td>
<td>+1+i0</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>+iY</td>
<td>-1+i0 Tan(Y)</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN</td>
<td>+1+i0 Tan(Y)</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>+i0</td>
<td>-1+i0</td>
<td>-0+i0</td>
<td>+0+i0</td>
<td>+1+i0</td>
<td>QNAN+i0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-i0</td>
<td>-1-i0</td>
<td>-0-i0</td>
<td>+0-i0</td>
<td>+1-i0</td>
<td>QNAN-i0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-iY</td>
<td>-1-i0 Tan(Y)</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN</td>
<td>+1+i0 Tan(Y)</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>-i∞</td>
<td>-1-i0</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN INVALID</td>
<td>QNAN+iQNAN</td>
<td>+1-i0</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>+iNAN</td>
<td>-1+i0</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>+1+i0</td>
<td>QNAN+iQNAN</td>
</tr>
</tbody>
</table>
Notes:
• raises INVALID exception when real or imaginary part of the argument is SNAN
• \( \text{Tanh}(\text{CONJ}(z)) = \text{CONJ}(\text{Tanh}(z)) \)
• \( \text{Tanh}(-z) = -\text{Tanh}(z) \).

\textbf{vAcosh}

\textit{Computes inverse hyperbolic cosine (nonnegative) of vector elements.}

\textbf{Syntax}

\begin{verbatim}
vsAcosh( n, a, y );
vmsAcosh( n, a, y, mode );
vdAcosh( n, a, y );
vmdAcosh( n, a, y, mode );
vcAcosh( n, a, y );
vmcAcosh( n, a, y, mode );
vzAcosh( n, a, y );
vmzAcosh( n, a, y, mode );
\end{verbatim}

\textbf{Include Files}

• mkl.h

\textbf{Input Parameters}

\begin{tabular}{|l|l|l|}
\hline
Name & Type & Description \\
\hline
\textit{n} & \text{const MKL\_INT} & Specifies the number of elements to be calculated. \\
\hline
\textit{a} & \text{const float* for vsAcosh,} & Pointer to an array that contains the input vector \textit{a}. \\
& \text{vmsAcosh} & \\
& \text{const double* for vdAcosh,} & \\
& \text{vmdAcosh} & \\
& \text{const MKL\_Complex8* for vcAcosh,} & \\
& \text{vmcAcosh} & \\
& \text{const MKL\_Complex16* for vzAcosh,} & \\
& \text{vmzAcosh} & \\
\hline
\textit{mode} & \text{const MKL\_INT64} & Overrides global VM mode setting for this function call. See \texttt{vmlSetMode} for possible values and their description. \\
\hline
\end{tabular}

\textbf{Output Parameters}

\begin{tabular}{|l|l|l|}
\hline
Name & Type & Description \\
\hline
\textit{y} & \text{float* for vsAcosh, vmsAcosh} & Pointer to an array that contains the output vector \textit{y}. \\
& \text{double* for vdAcosh, vmdAcosh} & \\
\hline
\end{tabular}
**Name** | **Type** | **Description**
---|---|---
MKL_Complex8* for vcAcosh, vmcAcosh
MKL_Complex16* for vzAcosh, vmzAcosh

**Description**
The `v?Acosh` function computes inverse hyperbolic cosine (nonnegative) of vector elements.

### Special Values for Real Function `v?Acosh(x)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X &lt; +1</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the [Special Value Notations](#) section for the conventions used in the table below.

### Special Values for Complex Function `v?Acosh(z)`

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+∞i</td>
<td>-∞+i 3π/4</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
</tr>
<tr>
<td>+i</td>
<td>+∞+iπ</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
</tr>
<tr>
<td>+i0</td>
<td>+∞+iπ</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
</tr>
<tr>
<td>-i0</td>
<td>+∞+iπ</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
</tr>
<tr>
<td>-i</td>
<td>+∞+iπ</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
<td>+∞+i0</td>
</tr>
<tr>
<td>-i∞</td>
<td>+∞+iπ</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
</tr>
<tr>
<td>+iNAN</td>
<td>+∞+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
</tbody>
</table>

**Notes:**
- raises INVALID exception when real or imaginary part of the argument is SNAN
- \( \text{Acosh} (\text{CONJ}(z)) = \text{CONJ} (\text{Acosh}(z)). ~ \)

### `v?Asinh`

**Computes inverse hyperbolic sine of vector elements.**

**Syntax**

vsAsinh( n, a, y );

vmsAsinh( n, a, y, mode );

vdAsinh( n, a, y );
vmdAsinh( n, a, y, mode );
vcaSinh( n, a, y );
vmcAsinh( n, a, y, mode );
vzAsinh( n, a, y );
vmzAsinh( n, a, y, mode );

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsAsinh, vmsAsinh const double* for vdAsinh, vmdAsinh const MKL_Complex8* for vcAsinh, vmcAsinh const MKL_Complex16* for vzAsinh, vmzAsinh</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsAsinh, vmsAsinh double* for vdAsinh, vmdAsinh MKL_Complex8* for vcAsinh, vmcAsinh MKL_Complex16* for vzAsinh, vmzAsinh</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description
The v?Asinh function computes inverse hyperbolic sine of vector elements.

Special Values for Real Function v?Asinh(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>Argument</td>
<td>Result</td>
<td>Exception</td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>-----------</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the Special Value Notations section for the conventions used in the table below.

### Special Values for Complex Function v?Asinh(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>i∞</td>
<td>+∞</td>
<td>-∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/2</td>
<td>+∞+iπ/4</td>
<td>+∞+iQNAN</td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+i</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
<td>+0+i0</td>
</tr>
<tr>
<td>-0</td>
<td>-i</td>
<td>-0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
<td>+0-i0</td>
</tr>
<tr>
<td>-i∞</td>
<td>-∞</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
<td>-∞-iπ/4</td>
</tr>
<tr>
<td>+i</td>
<td>+∞</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
</tr>
</tbody>
</table>

**Notes:**

- Raises INVALID exception when real or imaginary part of the argument is SNAN
- \( \text{Asinh(CONJ(z))}=\text{CONJ(Asinh(z))} \)
- \( \text{Asinh}(-z)=-\text{Asinh}(z) \)

**v?Atanh**

*Computes inverse hyperbolic tangent of vector elements.*

**Syntax**

```c
vsAtanh( n, a, y );
vmsAtanh( n, a, y, mode );
vdAtanh( n, a, y );
vmdAtanh( n, a, y, mode );
vcAtanh( n, a, y );
vmcAtanh( n, a, y, mode );
vzAtanh( n, a, y );
vmzAtanh( n, a, y, mode );
```

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>$a$</td>
<td>const float* for vsAtanh,</td>
<td>Pointer to an array that contains the input vector $a$.</td>
</tr>
<tr>
<td></td>
<td>vsAtanh, vmsAtanh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdAtanh,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmdAtanh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex8* for vcAtanh,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmcAtanh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16* for vzAtanh,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmzAtanh</td>
<td></td>
</tr>
<tr>
<td>$mode$</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsAtanh, vmsAtanh</td>
<td>Pointer to an array that contains the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>double* for vdAtanh, vmdAtanh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex8* for vcAtanh,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmcAtanh</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MKL_Complex16* for vzAtanh,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vmzAtanh</td>
<td></td>
</tr>
</tbody>
</table>

Description

The $v\text{?Atanh}$ function computes inverse hyperbolic tangent of vector elements.

Special Values for Real Function $v\text{?Atanh}(x)$

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+1$</td>
<td>$+\infty$</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>$-1$</td>
<td>$-\infty$</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>$</td>
<td>x</td>
<td>&gt; 1$</td>
<td>QNAN</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>$-\infty$</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See the Special Value Notations section for the conventions used in the table below.
Special Values for Complex Function v?Atanh(z)

<table>
<thead>
<tr>
<th>RE(z)</th>
<th>IM(z)</th>
<th>-∞</th>
<th>-X</th>
<th>-0</th>
<th>+0</th>
<th>+X</th>
<th>+∞</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>+i∞</td>
<td>-0+iπ/2</td>
<td>-0+iπ/2</td>
<td>-0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+∞</td>
<td>NaN+iQNAN</td>
</tr>
<tr>
<td>+iY</td>
<td>-0+iπ/2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+0+iπ/2</td>
<td>NaN+iQNAN</td>
<td></td>
</tr>
<tr>
<td>+i0</td>
<td>-0+iπ/2</td>
<td>-0+i0</td>
<td>+0+i0</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>NaN+iQNAN</td>
<td></td>
</tr>
<tr>
<td>-i0</td>
<td>-0-iπ/2</td>
<td>-0-i0</td>
<td>+0-i0</td>
<td>+0-iπ/2</td>
<td>+0-iπ/2</td>
<td>+0-iπ/2</td>
<td>NaN+iQNAN</td>
<td></td>
</tr>
<tr>
<td>-i∞</td>
<td>-0+iπ/2</td>
<td>-0+iπ/2</td>
<td>-0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0+iπ/2</td>
<td>+0-iπ/2</td>
<td>NaN+iQNAN</td>
</tr>
<tr>
<td>+iNAN</td>
<td>-0+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td>QNAN+iQNAN</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- Atanh(+1+i0)=+--∞,-i*0, and ZERODIVIDE exception is raised
- raises INVALID exception when real or imaginary part of the argument is SNAN
- Atanh(CONJ(z))=CONJ(Atanh(z))
- Atanh(-z)=-Atanh(z).

Special Functions

v?Erf
Computes the error function value of vector elements.

Syntax

```
vsErf( n, a, y );
vmsErf( n, a, y, mode );
vdErf( n, a, y );
vmdErf( n, a, y, mode );
```

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsErf, vmsErf const double* for vdErf, vmdErf</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsErf, vmsErf</td>
<td>Pointer to an array that contains the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>double* for vdErf, vmdErf</td>
<td></td>
</tr>
</tbody>
</table>

Description

The $Erf$ function computes the error function values for elements of the input vector $a$ and writes them to the output vector $y$.

The error function is defined as given by:

$$
erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.
$$

Useful relations:

1. $erfc(x) = 1 - erf(x)$,

   where $erfc$ is the complementary error function.

2. $\Phi(x) = \frac{1}{2} erf\left(\frac{x}{\sqrt{2}}\right)$,

   where

   $$
   \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp\left(-\frac{t^2}{2}\right) dt
   $$

   is the cumulative normal distribution function.

3. $\Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1)$,

   where $\Phi^{-1}(x)$ and $\text{erf}^{-1}(x)$ are the inverses to $\Phi(x)$ and $erf(x)$ respectively.

The following figure illustrates the relationships among $Erf$ family functions ($Erf$, $Erfc$, $CdfNorm$).
Useful relations for these functions:

\[ \text{erf}(x) + \text{erfc}(x) = 1 \]

\[ \text{cdfnorm}(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left( \frac{x}{\sqrt{2}} \right) \]

**Special Values for Real Function v?Erf(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+(\infty)</td>
<td>+1</td>
<td></td>
</tr>
<tr>
<td>-(\infty)</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**See Also**

Erfc
CdfNorm

v?Erfc

*Computes the complementary error function value of vector elements.*

**Syntax**

\[ \text{vsErfc}(n, a, y); \]
vmsErfc( n, a, y, mode );
vErfc( n, a, y );
vmdErfc( n, a, y, mode );

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsErfc, vmsErfc const double* for vdErfc, vmdErfc</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsErfc, vmsErfc const double* for vdErfc, vmdErfc</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The Erfc function computes the complementary error function values for elements of the input vector $a$ and writes them to the output vector $y$.

The complementary error function is defined as follows:

$$ \text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2} dt. $$

Useful relations:

1. $\text{erfc}(x) = 1 - \text{erf}(x)$.

2. $\Phi(x) = \frac{1}{2} \text{erf} \left( \frac{x}{\sqrt{2}} \right)$,

where

$$ \Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp \left( - \frac{t^2}{2} \right) dt $$

is the cumulative normal distribution function.
3. \( \Phi^{-1}(x) = \sqrt{2} \text{erf}^{-1}(2x - 1) \),

where \( \Phi^{-1}(x) \) and \( \text{erf}^{-1}(x) \) are the inverses to \( \Phi(x) \) and \( \text{erf}(x) \) respectively.

See also Figure "Erf Family Functions Relationship" in Erf function description for Erfc function relationship with the other functions of Erf family.

### Special Values for Real Function \( \text{v?Erfc}(x) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &gt; \text{underflow} )</td>
<td>+0</td>
<td>VML_STATUS_UNDERFLOW</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>( +\infty )</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( -\infty )</td>
<td>+2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**See Also**

Erf

CdfNorm

### \( \text{v?CdfNorm} \)

*Computes the cumulative normal distribution function values of vector elements.*

**Syntax**

```c
vsCdfNorm( n, a, y );
```

```c
vmsCdfNorm( n, a, y, mode );
```

```c
vdCdfNorm( n, a, y );
```

```c
vmdCdfNorm( n, a, y, mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a )</td>
<td>const float* for vsCdfNorm, vmsCdfNorm const double* for vdCdfNorm, vmdCdfNorm</td>
<td>Pointer to an array that contains the input vector ( a ).</td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>float* for vsCdfNorm, vmsCdfNorm</td>
<td>Pointer to an array that contains the output vector $y$.</td>
</tr>
<tr>
<td></td>
<td>double* for vdCdfNorm, vmdCdfNorm</td>
<td></td>
</tr>
</tbody>
</table>

Description

The $\text{CdfNorm}$ function computes the cumulative normal distribution function values for elements of the input vector $a$ and writes them to the output vector $y$.

The cumulative normal distribution function is defined as given by:

$$
\text{CdfNorm}(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{t^2}{2}} dt.
$$

Useful relations:

$$
\text{cdfnorm}(x) = \frac{1}{2} \left( 1 + \text{erf} \left( \frac{x}{\sqrt{2}} \right) \right) = 1 - \frac{1}{2} \text{erfc} \left( \frac{x}{\sqrt{2}} \right).
$$

where $\text{Erf}$ and $\text{Erfc}$ are the error and complementary error functions.

See also Figure "Erf Family Functions Relationship" in Erf function description for CdfNorm function relationship with the other functions of Erf family.

Special Values for Real Function $v?\text{CdfNorm}(x)$

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt; $underflow</td>
<td>+0</td>
<td>VML_STATUS_UNDERFLOW</td>
<td>UNDERFLOW</td>
</tr>
<tr>
<td>$+\infty$</td>
<td>+1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$-\infty$</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See Also

Erf
Erfc

$v?\text{ErfInv}$

Computes inverse error function value of vector elements.

Syntax

```c
vsErfInv( n, a, y );

vmsErfInv( n, a, y, mode );

vdErfInv( n, a, y );

vmdErfInv( n, a, y, mode );
```
Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsErfInv, vmsErfInv const double* for vdErfInv, vmdErfInv</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsErfInv, vmsErfInv double* for vdErfInv, vmdErfInv</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The ErfInv function computes the inverse error function values for elements of the input vector a and writes them to the output vector y.

\[ y = \text{erf}^{-1}(a), \]

where \( \text{erf}(x) \) is the error function defined as given by:

\[
\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt.
\]

Useful relations:

1. \( \text{erf}^{-1}(x) = \text{erfc}^{-1}(1 - x) \),

where \( \text{erfc} \) is the complementary error function.

2. \( \Phi(x) = \frac{1}{2} \text{erf} \left( \frac{x}{\sqrt{2}} \right) \),

where

\[
\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \exp \left( - \frac{t^2}{2} \right) dt
\]

is the cumulative normal distribution function.
3. \( \Phi^{-1}(x) = \sqrt{2} \operatorname{erf}^{-1}(2x - 1) \),

where \( \Phi^{-1}(x) \) and \( \operatorname{erf}^{-1}(x) \) are the inverses to \( \Phi(x) \) and \( \operatorname{erf}(x) \) respectively.

Figure "ErfInv Family Functions Relationship" illustrates the relationships among \( \text{ErfInv} \) family functions (\( \text{ErfInv}, \text{ErfcInv}, \text{CdfNormInv} \)).

Useful relations for these functions:

\[
\operatorname{erfcinv}(x) = \operatorname{erfinv}(1 - x)
\]

\[
\text{CdfNormInv}(x) = \sqrt{2} \operatorname{erfinv}(2x - 1) = \sqrt{2} \operatorname{erfcinv}(2 - 2x)
\]

**Special Values for Real Function v?ErfInv(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+1</td>
<td>+\infty</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-1</td>
<td>-\infty</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>(</td>
<td>X</td>
<td>&gt; 1</td>
<td>QNAN</td>
</tr>
<tr>
<td>+\infty</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-\infty</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The `ErfcInv` function computes the inverse complimentary error function values for elements of the input vector `a` and writes them to the output vector `y`.

The inverse complementary error function is defined as given by:

\[ \text{erfcinv}(x) = \text{erfinv}(1 - x) \]
\[ \text{erfinv}(x) = \text{erf}^{-1}(x) \]

\[ \text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2} dt \]

where \( \text{erf}(x) \) denotes the error function and \( \text{erfinv}(x) \) denotes the inverse error function.

See also Figure "ErfInv Family Functions Relationship" in ErfInv function description for ErfcInv function relationship with the other functions of ErfInv family.

### Special Values for Real Function \( \text{v?ErfcInv}(x) \)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td>(-\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>(+\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>(+\infty)</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>(X &lt; -0)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(X &gt; +2)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(+\infty)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>(-\infty)</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

See Also
ErfInv
CdfNormInv

\( \text{v?CdfNormInv} \)
Computes the inverse cumulative normal distribution function values of vector elements.

### Syntax

- \( \text{vsCdfNormInv}( n, a, y ); \)
- \( \text{vmsCdfNormInv}( n, a, y, \text{mode} ); \)
- \( \text{vdCdfNormInv}( n, a, y ); \)
- \( \text{vmdCdfNormInv}( n, a, y, \text{mode} ); \)

### Include Files
- \( \text{mkl.h} \)

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* ( \text{for vsCdfNormInv, vmsCdfNormInv} ),</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
</tbody>
</table>
### Description

The `CdfNormInv` function computes the inverse cumulative normal distribution function values for elements of the input vector `x` and writes them to the output vector `y`.

The inverse cumulative normal distribution function is defined as given by:

\[
CdfNormInv(x) = CdfNorm^{-1}(x)
\]

where `CdfNorm(x)` denotes the cumulative normal distribution function.

Useful relations:

\[
cdfNormInv(x) = \sqrt{2} \text{erfinv}(2x - 1) = \sqrt{2} \text{erfcinv}(2 - 2x)
\]

where `erfinv(x)` denotes the inverse error function and `erfcinv(x)` denotes the inverse complementary error functions.

See also Figure "ErfInv Family Functions Relationship" in `ErfInv` function description for `CdfNormInv` function relationship with the other functions of `ErfInv` family.

### Special Values for Real Function `v?CdfNormInv(x)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.5</td>
<td>+0</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+1</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>+0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>X &lt; -0</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>X &gt; +1</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
</tbody>
</table>

### See Also

`ErfInv`
ErfcInv

v?LGamma

Computes the natural logarithm of the absolute value of gamma function for vector elements.

Syntax

vsLGamma( n, a, y );
vmsLGamma( n, a, y, mode );
vdLGamma( n, a, y );
vmdLGamma( n, a, y, mode );

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsLGamma, vmsLGamma</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdLGamma, vmdLGamma</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsLGamma, vmsLGamma</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdLGamma, vmdLGamma</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?LGamma function computes the natural logarithm of the absolute value of gamma function for elements of the input vector a and writes them to the output vector y. Precision overflow thresholds for the v?LGamma function are beyond the scope of this document. If the result does not meet the target precision, the function raises the OVERFLOW exception and sets the VM Error Status to VML_STATUS_OVERFLOW.

Special Values for Real Function v?LGamma(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+2</td>
<td>+0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>-0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>negative integer</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
</tbody>
</table>
### Argument Result VM Error Status Exception

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>-(\infty)</td>
<td>+(\infty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+(\infty)</td>
<td>+(\infty)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(X &gt; \text{overflow})</td>
<td>+(\infty)</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### v?TGamma

*Computes the gamma function of vector elements.*

### Syntax

- \(\text{vsTGamma}( n, a, y );\)
- \(\text{vmsTGamma}( n, a, y, \text{mode} );\)
- \(\text{vdTGamma}( n, a, y );\)
- \(\text{vmdTGamma}( n, a, y, \text{mode} );\)

### Include Files

- mkl.h

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float*</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
<tr>
<td>(\text{mode})</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See (\text{vmlSetMode}) for possible values and their description.</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsTGamma, vmsTGamma</td>
<td>Pointer to an array that contains the output vector (y).</td>
</tr>
<tr>
<td></td>
<td>double* for vdTGamma, vmdTGamma</td>
<td></td>
</tr>
</tbody>
</table>

### Description

The \(\text{v?TGamma}\) function computes the gamma function for elements of the input vector \(a\) and writes them to the output vector \(y\). Precision overflow thresholds for the \(\text{v?TGamma}\) function are beyond the scope of this document. If the result does not meet the target precision, the function raises the \(\text{OVERFLOW}\) exception and sets the VM Error Status to VML_STATUS_OVERFLOW.
### Special Values for Real Function vTGamma(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVE</td>
</tr>
<tr>
<td>-0</td>
<td>-∞</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVE</td>
</tr>
<tr>
<td>negative integer</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>-∞</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>X &gt; overflow</td>
<td>+∞</td>
<td>VML_STATUS_OVERFLOW</td>
<td>OVERFLOW</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### vExpInt1

*Computes the exponential integral of vector elements.*

**Syntax**

```c
vsExpInt1( n, a, y );
vmsExpInt1( n, a, y, mode );
vdExpInt1( n, a, y );
vmdExpInt1( n, a, y, mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsExpInt1, vmsExpInt1</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdExpInt1, vmdExpInt1</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsExpInt1, vmsExpInt1</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdExpInt1, vmdExpInt1</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The `vExpInt1` function computes the exponential integral $E_1$ of vector elements.

For positive real values $x$, this can be written as:
\[ E_i(x) = \int_x^\infty \frac{e^{-t}}{t} \, dt = \int_1^\infty \frac{e^{-xt}}{t} \, dt. \]

For negative real values \( x \), the result is defined as \text{NAN}.

**Special Values for Real Function \texttt{v?ExpInt1}(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x &lt; +0 )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>( +0 )</td>
<td>+( \infty )</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>( -0 )</td>
<td>+( \infty )</td>
<td>VML_STATUS_SING</td>
<td>ZERODIVIDE</td>
</tr>
<tr>
<td>( +\infty )</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( -\infty )</td>
<td>QNAN</td>
<td>VML_STATUS_ERRDOM</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Rounding Functions**

**\texttt{v?Floor}**

*Computes an integer value rounded towards minus infinity for each vector element.*

**Syntax**

\[
\begin{align*}
\text{vsFloor}( n, a, y ); \\
\text{vmsFloor}( n, a, y, mode ); \\
\text{vdFloor}( n, a, y ); \\
\text{vmdFloor}( n, a, y, mode );
\end{align*}
\]

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
| \( a \) | const float* for \texttt{vsFloor}, \texttt{vmsFloor}  \\
|        | const double* for \texttt{vdFloor}, \texttt{vmdFloor} | Pointer to an array that contains the input vector \( a \).                 |
| \( mode \) | const MKL_INT64 | Overrides global VM mode setting for this function call. See \texttt{vmlSetMode} for possible values and their description. |
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsFloor, vmsFloor</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdFloor, vmdFloor</td>
<td></td>
</tr>
</tbody>
</table>

Description

The function computes an integer value rounded towards minus infinity for each vector element.

\[ y_i = \lfloor a_i \rfloor \]

Special Values for Real Function vFloor(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

vCeil

*Computes an integer value rounded towards plus infinity for each vector element.*

Syntax

vsCeil( n, a, y);
vmsCeil( n, a, y, mode );
vdCeil( n, a, y);
vmdCeil( n, a, y, mode );

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsCeil, vmsCeil</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td></td>
<td>const double* for vdCeil, vmdCeil</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsCeil, vmsCeil; double* for vdCeil, vmdCeil</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description

The function computes an integer value rounded towards plus infinity for each vector element.

\[
y_i = \lceil a_i \rceil
\]

Special Values for Real Function v?Ceil(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+\infty</td>
<td>+\infty</td>
<td></td>
</tr>
<tr>
<td>-\infty</td>
<td>-\infty</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

v?Trunc

Computes an integer value rounded towards zero for each vector element.

Syntax

vsTrunc( n, a, y );
vmsTrunc( n, a, y, mode );
vmdTrunc( n, a, y );
vdTrunc( n, a, y );

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsTrunc, vmsTrunc; const double* for vdTrunc, vmdTrunc</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsTrunc, vmsTrunc</td>
<td>Pointer to an array that contains the output vector (y).</td>
</tr>
<tr>
<td></td>
<td>double* for vdTrunc, vmdTrunc</td>
<td></td>
</tr>
</tbody>
</table>

Description

The function computes an integer value rounded towards zero for each vector element.

\[
a_i \geq 0, y_i = \lfloor a_i \rfloor \\
\lfloor a_i \rfloor \\
\]

Special Values for Real Function v?Trunc(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+(\infty)</td>
<td>+(\infty)</td>
<td></td>
</tr>
<tr>
<td>-(\infty)</td>
<td>-(\infty)</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

v?Round

*Computes a value rounded to the nearest integer for each vector element.*

Syntax

```
vsRound( n, a, y );
vmsRound( n, a, y, mode );
vdRound( n, a, y );
vmdRound( n, a, y, mode );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsRound, vmsRound</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdRound, vmdRound</td>
<td>Pointer to an array that contains the input vector (a).</td>
</tr>
</tbody>
</table>
### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsRound, vmsRound double* for vdRound, vmdRound</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

### Description

The function computes a value rounded to the nearest integer for each vector element. Input elements that are halfway between two consecutive integers are always rounded away from zero regardless of the rounding mode.

### Special Values for Real Function v?Round(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

### v?NearbyInt

*Computes a rounded integer value in the current rounding mode for each vector element.*

#### Syntax

```c
vsNearbyInt( n, a, y );
vmsNearbyInt( n, a, y, mode );
vdNearbyInt( n, a, y );
vmdNearbyInt( n, a, y, mode );
```

#### Include Files

- mkl.h

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsNearbyInt, vmsNearbyInt</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>const double*</td>
<td>for vdNearbyInt, vmdNearbyInt</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsNearbyInt, vmsNearbyInt</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
<tr>
<td></td>
<td>double* for vdNearbyInt, vmdNearbyInt</td>
<td></td>
</tr>
</tbody>
</table>

### Description

The v?NearbyInt function computes a rounded integer value in a current rounding mode for each vector element.

### Special Values for Real Function v?NearbyInt(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

### v?Rint

*Computes a rounded integer value in the current rounding mode.*

### Syntax

```c
vsRint( n, a, y );
vmsRint( n, a, y, mode );
vdRint( n, a, y );
vmdRint( n, a, y, mode );
```

### Include Files

- mkl.h

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
\(a\) | const float* for vsRint, vmsRint, const double* for vdRint, vmdRint | Pointer to an array that contains the input vector \(a\).
\(mode\) | const MKL_INT64 | Overrides global VM mode setting for this function call. See \texttt{vmlSetMode} for possible values and their description.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| \(y\) | float* for vsRint, vmsRint, double* for vdRint, vmdRint | Pointer to an array that contains the output vector \(y\).

**Description**

The \texttt{vRint} function computes a rounded floating-point integer value using the current rounding mode for each vector element.

The rounding mode affects the results computed for inputs that fall between consecutive integers. For example:

- \(f(0.5) = 0\), for rounding modes set to round to nearest round toward zero or to minus infinity.
- \(f(0.5) = 1\), for rounding modes set to plus infinity.
- \(f(-1.5) = -2\), for rounding modes set to round to nearest or to minus infinity.
- \(f(-1.5) = -1\), for rounding modes set to round toward zero or to plus infinity.

**Special Values for Real Function vRint(x)**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+(\infty)</td>
<td>+(\infty)</td>
<td></td>
</tr>
<tr>
<td>-(\infty)</td>
<td>-(\infty)</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

\texttt{v?Modf}

*Computes a truncated integer value and the remaining fraction part for each vector element.*

**Syntax**

\begin{verbatim}
vsModf( n, a, y, z );
vmsModf( n, a, y, z, mode );
vdModf( n, a, y, z );
vmdModf( n, a, y, z, mode );
\end{verbatim}

**Include Files**

- \texttt{mkl.h}
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsModf, vmsModf, const double* for vdModf, vmdModf</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y, z</td>
<td>float* for vsModf, vmsModf, double* for vdModf, vmdModf</td>
<td>Pointer to an array that contains the output vector y and z.</td>
</tr>
</tbody>
</table>

Description

The function computes a truncated integer value and the remaining fraction part for each vector element.

\[
\begin{align*}
& a_i \geq 0, \quad \begin{cases} y_i = \lfloor a_i \rfloor \\ z_i = a_i - \lfloor a_i \rfloor \end{cases} \\
& a_i < 0, \quad \begin{cases} y_i = \lceil a_i \rceil \\ z_i = a_i - \lceil a_i \rceil \end{cases}
\end{align*}
\]

Special Values for Real Function v?Modf(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result: y(i)</th>
<th>Result: z(i)</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+∞</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-∞</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

v?Frac

*Computes a signed fractional part for each vector element.*

Syntax

```
vsFrac( n, a, y );
vmsFrac( n, a, y, mode );
vdFrac( n, a, y );
vmdFrac( n, a, y, mode );
```
Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float* for vsFrac, vmsFrac, const double* for vdFrac, vmdFrac</td>
<td>Pointer to an array that contains the input vector a.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsFrac, vmsFrac, double* for vdFrac, vmdFrac</td>
<td>Pointer to an array that contains the output vector y.</td>
</tr>
</tbody>
</table>

Description
The function computes a signed fractional part for each vector element.

\[ y_i = \begin{cases} a_i - \lfloor a_i \rfloor, & a_i \geq 0 \\ a_i - \lceil a_i \rceil, & a_i < 0 \end{cases} \]

Special Values for Real Function v?Frac(x)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Result</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-0</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>+∞</td>
<td>+0</td>
<td></td>
</tr>
<tr>
<td>-∞</td>
<td>-0</td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>QNAN</td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
</tr>
</tbody>
</table>

VM Pack/Unpack Functions

This section describes VM functions that convert vectors with unit increment to and from vectors with positive increment indexing, vector indexing, and mask indexing (see Appendix B for details on vector indexing methods).

The table below lists available VM Pack/Unpack functions, together with data types and indexing methods associated with them.
VM Pack/Unpack Functions

<table>
<thead>
<tr>
<th>Function Short Name</th>
<th>Data Types</th>
<th>Indexing Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>v?Pack</td>
<td>s, d, c, z</td>
<td>I, V, M</td>
<td>Gathers elements of arrays, indexed by different methods.</td>
</tr>
<tr>
<td>v?Unpack</td>
<td>s, d, c, z</td>
<td>I, V, M</td>
<td>Scatters vector elements to arrays with different indexing.</td>
</tr>
</tbody>
</table>

See Also
Appendix B

v?Pack
Copies elements of an array with specified indexing to a vector with unit increment.

Syntax
vsPackI( n, a, inca, y );
vsPackV( n, a, ia, y );
vsPackM( n, a, ma, y );
vdPackI( n, a, inca, y );
vPackV( n, a, ia, y );
vPackM( n, a, ma, y );
vcPackI( n, a, inca, y );
vPackV( n, a, ia, y );
vPackM( n, a, ma, y );
vzPackI( n, a, inca, y );
vzPackV( n, a, ia, y );
vzPackM( n, a, ma, y );

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
</tbody>
</table>
| a    | const float*: for vsPackI, vsPackV, vsPackM  
      |       | Specifies pointer to an array that contains the input vector a. The arrays must be: |
      |       | for v?PackI, at least \((1 + (n-1)*inca)\) |
      |       | for v?PackV, at least \(\max(n, \max(ia[j]) \), j=0, ... , n-1\) |
      |       | for v?PackM, at least \(n\) |
### v?Unpack

Copies elements of a vector with unit increment to an array with specified indexing.

#### Syntax

vsUnpackI( n, a, y, incy );  
vsUnpackV( n, a, y, iy );  
vsUnpackM( n, a, y, my );  
vUnpackI( n, a, y, incy );  
vUnpackV( n, a, y, iy );  
vUnpackM( n, a, y, my );  
vcUnpackI( n, a, y, incy );  
vcUnpackV( n, a, y, iy );  
vcUnpackM( n, a, y, my );  
vzUnpackI( n, a, y, incy );  
vzUnpackV( n, a, y, iy );  
vzUnpackM( n, a, y, my );

#### Include Files

- mkl.h
### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n)</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>(a)</td>
<td>const float* for vsUnpackI, vsUnpackV, vsUnpackM  &lt;br&gt; const double* for vdUnpackI, vdUnpackV, vdUnpackM  &lt;br&gt; const MKL_Complex8* for &lt;br&gt; vcUnpackI, vcUnpackV, vcUnpackM  &lt;br&gt; const MKL_Complex16* for &lt;br&gt; vzUnpackI, vzUnpackV, vzUnpackM</td>
<td>Specifies the pointer to an array of size at least (n) that contains the input vector (a).</td>
</tr>
<tr>
<td>(incy)</td>
<td>const MKL_INT for vsUnpackI, vdUnpackI, vcUnpackI, vzUnpackI</td>
<td>Specifies the increment for the elements of (y).</td>
</tr>
<tr>
<td>(iy)</td>
<td>const int* for vsUnpackV, vdUnpackV, vcUnpackV, vzUnpackV</td>
<td>Specifies the pointer to an array of size at least (n) that contains the index vector for the elements of (a).</td>
</tr>
<tr>
<td>(my)</td>
<td>const int* for vsUnpackM, vdUnpackM, vcUnpackM, vzUnpackM</td>
<td>Specifies the pointer to an array of size at least (n) that contains the mask vector for the elements of (a).</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(y)</td>
<td>float* for vsUnpackI, vsUnpackV, vsUnpackM  &lt;br&gt; double* for vdUnpackI, vdUnpackV, vdUnpackM  &lt;br&gt; const MKL_Complex8* for &lt;br&gt; vcUnpackI, vcUnpackV, vcUnpackM  &lt;br&gt; const MKL_Complex16* for &lt;br&gt; vzUnpackI, vzUnpackV, vzUnpackM</td>
<td>Specifies the pointer to an array that contains the output vector (y).  &lt;br&gt; The array must be:  &lt;br&gt; for (v?\text{UnpackI}), at least ((1 + (n-1)\times incy))  &lt;br&gt; for (v?\text{UnpackV}), at least (\max( n, \max(ia[j]) )), (j=0,\ldots, n-1)  &lt;br&gt; for (v?\text{UnpackM}), at least (n).</td>
</tr>
</tbody>
</table>

### VM Service Functions

The VM Service functions enable you to set/get the accuracy mode and error code. These functions are available both in the Fortran and C interfaces. The table below lists available VM Service functions and their short description.

<table>
<thead>
<tr>
<th>VM Service Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vmlSetMode</td>
<td>Sets the VM mode</td>
</tr>
<tr>
<td>vmlGetMode</td>
<td>Gets the VM mode</td>
</tr>
</tbody>
</table>
### Function Short Name | Description
---|---
MKLFreeTls | Frees allocated VM/VS thread local storage memory from within DllMain routine (Windows* OS only)
vmlSetErrStatus | Sets the VM Error Status
vmlGetErrStatus | Gets the VM Error Status
vmlClearErrStatus | Clears the VM Error Status
vmlSetErrorCallBack | Sets the additional error handler callback function
vmlGetErrorCallBack | Gets the additional error handler callback function
vmlClearErrorCallBack | Deletes the additional error handler callback function

---

**vmlSetMode**

Sets a new mode for VM functions according to the `mode` parameter and stores the previous VM mode to `oldmode`.

**Syntax**

```c
oldmode = vmlSetMode( mode );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mode</code></td>
<td>const MKL_UINT</td>
<td>Specifies the VM mode to be set.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>oldmode</code></td>
<td>unsigned int</td>
<td>Specifies the former VM mode.</td>
</tr>
</tbody>
</table>

**Description**

The `vmlSetMode` function sets a new mode for VM functions according to the `mode` parameter and stores the previous VM mode to `oldmode`. The mode change has a global effect on all the VM functions within a thread.

**NOTE**

You can override the global mode setting and change the mode for a given VM function call by using a respective `vml[<s,d]<Func>` variant of the function.

The `mode` parameter is designed to control accuracy, handling of denormalized numbers, and error handling. Table "Values of the `mode` Parameter" lists values of the `mode` parameter. You can obtain all other possible values of the `mode` parameter from the `mode` parameter values by using bitwise OR ( | ) operation to combine one value for accuracy, one value for handling of denormalized numbers, and one value for error control options. The default value of the `mode` parameter is `VML_HA | VML_FTZDAZ_OFF | VML_ERRMODE_DEFAULT`. 

---

2009
The VML_FTZDAZ_ON mode is specifically designed to improve the performance of computations that involve denormalized numbers at the cost of reasonable accuracy loss. This mode changes the numeric behavior of the functions: denormalized input values are treated as zeros (DAZ = denormals-are-zero) and denormalized results are flushed to zero (FTZ = flush-to-zero). Accuracy loss may occur if input and/or output values are close to denormal range.

### Values of the mode Parameter

<table>
<thead>
<tr>
<th>Value of mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy Control</td>
<td></td>
</tr>
<tr>
<td>VML_HA</td>
<td>high accuracy versions of VM functions</td>
</tr>
<tr>
<td>VML_LA</td>
<td>low accuracy versions of VM functions</td>
</tr>
<tr>
<td>VML_EP</td>
<td>enhanced performance accuracy versions of VM functions</td>
</tr>
<tr>
<td>Denormalized Numbers Handling Control</td>
<td></td>
</tr>
<tr>
<td>VML_FTZDAZ_ON</td>
<td>Faster processing of denormalized inputs is enabled.</td>
</tr>
<tr>
<td>VML_FTZDAZ_OFF</td>
<td>Faster processing of denormalized inputs is disabled.</td>
</tr>
<tr>
<td>Error Mode Control</td>
<td></td>
</tr>
<tr>
<td>VML_ERRMODE_IGNORE</td>
<td>On computation error, VM Error status is updated, but otherwise no action is set. Cannot be combined with other VML_ERRMODE settings.</td>
</tr>
<tr>
<td>VML_ERRMODE_NOERR</td>
<td>On computation error, VM Error status is not updated and no action is set. Cannot be combined with other VML_ERRMODE settings.</td>
</tr>
<tr>
<td>VML_ERRMODE_ERRNO</td>
<td>On error, the errno variable is set.</td>
</tr>
<tr>
<td>VML_ERRMODE_STDERR</td>
<td>On error, the error text information is written to stderr.</td>
</tr>
<tr>
<td>VML_ERRMODE_EXCEPT</td>
<td>On error, an exception is raised.</td>
</tr>
<tr>
<td>VML_ERRMODE_CALLBACK</td>
<td>On error, an additional error handler function is called.</td>
</tr>
<tr>
<td>VML_ERRMODE_DEFAULT</td>
<td>On error, the errno variable is set, an exception is raised, and an additional error handler function is called.</td>
</tr>
</tbody>
</table>

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Examples

The following example shows how to set low accuracy, fast processing for denormalized numbers and stderr error mode:

```c
vmlSetMode( VML_LA );
vmlSetMode( VML_LA | VML_FTZDAZ_ON | VML_ERRMODE_STDERR );
```

### vmlGetMode

Gets the VM mode.
Syntax

```c
mod = vmlGetMode( void);
```

Include Files

- mkl.h

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mod</td>
<td>unsigned int</td>
<td>Specifies the packed mode parameter.</td>
</tr>
</tbody>
</table>

Description

The function `vmlGetMode` returns the VM `mode` parameter that controls accuracy, handling of denormalized numbers, and error handling options. The `mod` variable value is a combination of the values listed in the table "Values of the mode Parameter". You can obtain these values using the respective mask from the table "Values of Mask for the mode Parameter".

Values of Mask for the mode Parameter

<table>
<thead>
<tr>
<th>Value of mask</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VML_ACCURACY_MASK</td>
<td>Specifies mask for accuracy mode selection.</td>
</tr>
<tr>
<td>VML_FTZDAZ_MASK</td>
<td>Specifies mask for FTZDAZ mode selection.</td>
</tr>
<tr>
<td>VML_ERRMODE_MASK</td>
<td>Specifies mask for error mode selection.</td>
</tr>
</tbody>
</table>

See example below:

Examples

```c
accm = vmlGetMode( void ) & VML_ACCURACY_MASK;
denm = vmlGetMode( void ) & VML_FTZDAZ_MASK;
errm = vmlGetMode( void ) & VML_ERRMODE_MASK;
```

MKLFreeTls

Frees allocated VM/VS thread local storage memory from within DllMain routine. Use on Windows* OS only.

Syntax

```c
void MKLFreeTls( const MKL_UINT fdwReason );
```

Include Files

- mkl_vml_functions_win.h

Description

The `MKLFreeTls` routine frees thread local storage (TLS) memory which has been allocated when using MKL static libraries to link into a DLL on the Windows* OS. The routine should only be used within DllMain.

NOTE

It is only necessary to use `MKLFreeTls` for TLS data in the VM and VS domains.
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdwReason</td>
<td>const MKL_UINT</td>
<td>Reason code from the DllMain call.</td>
</tr>
</tbody>
</table>

Example

```c
BOOL WINAPI DllMain(HINSTANCE hInst, DWORD fdwReason, LPVOID lpvReserved)
{
    /*customer code*/
    MKLFreeTls(fdwReason);
    /*customer code*/
}
```

vmlSetErrStatus

Sets the new VM Error Status according to err and stores the previous VM Error Status to olderrSets the global VM Status according to new values and returns the previous VM Status.

Syntax

```c
olderr = vmlSetErrStatus( status );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>const MKL_INT</td>
<td>Specifies the VM error status to be set.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olderr</td>
<td>int</td>
<td>Specifies the former VM error status.</td>
</tr>
</tbody>
</table>

Description

Table "Values of the VM Status" lists possible values of the err parameter.

Values of the VM Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Successful Execution</strong></td>
<td></td>
</tr>
<tr>
<td>VML_STATUS_OK</td>
<td>The execution was completed successfully.</td>
</tr>
<tr>
<td><strong>Warnings</strong></td>
<td></td>
</tr>
<tr>
<td>VML_STATUS_ACCURACYWARNING</td>
<td>The execution was completed successfully in a different accuracy mode.</td>
</tr>
<tr>
<td><strong>Errors</strong></td>
<td></td>
</tr>
</tbody>
</table>
### Status

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VML_STATUS_BADSIZE</td>
<td>The function does not support the preset accuracy mode. The Low Accuracy mode is used instead.</td>
</tr>
<tr>
<td>VML_STATUS_BADMEM</td>
<td>NULL pointer is passed.</td>
</tr>
<tr>
<td>VML_STATUS_ERRDOM</td>
<td>At least one of array values is out of a range of definition.</td>
</tr>
<tr>
<td>VML_STATUS_SING</td>
<td>At least one of the input array values causes a divide-by-zero exception or produces an invalid (QNaN) result.</td>
</tr>
<tr>
<td>VML_STATUS_OVERFLOW</td>
<td>An overflow has happened during the calculation process.</td>
</tr>
<tr>
<td>VML_STATUS_UNDERFLOW</td>
<td>An underflow has happened during the calculation process.</td>
</tr>
</tbody>
</table>

### Examples

```c
olderr = vmlSetErrStatus(VML_STATUS_OK);
olderr = vmlSetErrStatus(VML_STATUS_ERRDOM);
olderr = vmlSetErrStatus(VML_STATUS_UNDERFLOW);
```

### vmlGetErrStatus

*Gets the VM Error Status.*

#### Syntax

```c
err = vmlGetErrStatus(void);
```

#### Include Files

- mkl.h

#### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>err</td>
<td>int</td>
<td>Specifies the VM error status.</td>
</tr>
</tbody>
</table>

### vmlClearErrStatus

*Sets the VM Error Status to VML_STATUS_OK and stores the previous VM Error Status to olderr.*

#### Syntax

```c
olderr = vmlClearErrStatus(void);
```

#### Include Files

- mkl.h

#### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>olderr</td>
<td>int</td>
<td>Specifies the former VM error status.</td>
</tr>
</tbody>
</table>
vmlSetErrorCallBack
Sets the additional error handler callback function and gets the old callback function.

Syntax
oldcallback = vmlSetErrorCallBack( callback );

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>Pointer to the callback function.</td>
</tr>
</tbody>
</table>

The callback function has the following format:

```c
static int __stdcall
MyHandler(DefVmlErrorContext* pContext)
{
    /* Handler body */
};
```
The passed error structure is defined as follows:

```c
typedef struct _DefVmlErrorContext {
    int iCode; /* Error status value */
    int iIndex; /* Index for bad array element, or bad array dimension, or bad array pointer */
    double dbA1; /* Error argument 1 */
    double dbA2; /* Error argument 2 */
    double dbR1; /* Error result 1 */
    double dbR2; /* Error result 2 */
    char cFuncName[64]; /* Function name */
    int iFuncNameLen; /* Length of function name */
    double dbA1Im; /* Error argument 1, imag part */
    double dbA2Im; /* Error argument 2, imag part */
    double dbR1Im; /* Error result 1, imag part */
    double dbR2Im; /* Error result 2, imag part */
} DefVmlErrorContext;
```

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldcallback</td>
<td>int</td>
<td>Pointer to the former callback function.</td>
</tr>
</tbody>
</table>

### Description

The callback function is called on each VM mathematical function error if VML_ERRMODE_CALLBACK error mode is set (see "Values of the mode Parameter").

Use the `vmlSetErrorCallBack()` function if you need to define your own callback function instead of default empty callback function.

The input structure for a callback function contains the following information about the error encountered:

- the input value that caused an error
- location (array index) of this value
- the computed result value
- error code
• name of the function in which the error occurred.

You can insert your own error processing into the callback function. This may include correcting the passed result values in order to pass them back and resume computation. The standard error handler is called after the callback function only if it returns 0.

**vmlGetErrorCallBack**

*Gets the additional error handler callback function.*

**Syntax**

```c
callback = vmlGetErrorCallBack( void );
```

**Include Files**

- `mkl.h`

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>Pointer to the callback function</td>
<td></td>
</tr>
</tbody>
</table>

**vmlClearErrorCallBack**

*Deletes the additional error handler callback function and retrieves the former callback function.*

**Syntax**

```c
oldcallback = vmlClearErrorCallBack( void );
```

**Include Files**

- `mkl.h`

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldcallback</td>
<td>int</td>
<td>Pointer to the former callback function</td>
</tr>
</tbody>
</table>

**Miscellaneous VM Functions**

**v?CopySign**

*Returns vector of elements of one argument with signs changed to match other argument elements.*

**Syntax**

```c
vsCopySign(n, a, y);
vmsCopySign(n, a, y, mode);
vdcCopySign(n, a, y);
```
vmdCopySign (n, a, y, mode);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Pointer to the array containing the input vector a.</td>
</tr>
<tr>
<td></td>
<td>for vsCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmsCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmdCopySign</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float*</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
<tr>
<td></td>
<td>for vsCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmsCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdCopySign</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmdCopySign</td>
<td></td>
</tr>
</tbody>
</table>

Description
The v?CopySign function returns the first vector argument elements with the sign changed to match the sign of the second vector argument's corresponding elements.

v?NextAfter
Returns vector of elements containing the next representable floating-point values following the values from the elements of one vector in the direction of the corresponding elements of another vector.

Syntax
vsNextAfter (n, a, b, y);

Include Files
• mkl.h
The `v?NextAfter` function returns a vector containing the next representable floating-point values following the first vector argument elements in the direction of the second vector argument's corresponding elements.

**Special cases:**

**Overflow**

The function raises overflow and inexact floating-point exceptions and sets `VML_STATUS_OVERFLOW` if an input vector argument element is finite and the corresponding result vector element value is infinite.

**Underflow**

The function raises underflow and inexact floating-point exceptions and sets `VML_STATUS_UNDERFLOW` if a result vector element value is subnormal or zero, and different from the corresponding input vector argument element.

Even though underflow or overflow can occur, the returned value is independent of the current rounding direction mode.

### v?Fdim

*Returns vector containing the differences of the corresponding elements of the vector arguments if the first is larger and +0 otherwise.*

**Syntax**

```c
vsFdim (n, a, b, y);
vmsFdim (n, a, b, y, mode);
```
vdFdim \( (n, a, b, y); \)

vmdFdim \( (n, a, b, y, mode); \)

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a, b )</td>
<td>const float* for vsFdim</td>
<td>Pointers to the arrays containing the input vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsFdim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdFdim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdFdim</td>
<td></td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM ( mode ) setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float* for vsFdim</td>
<td>Pointer to an array containing the output vector ( y ).</td>
</tr>
<tr>
<td></td>
<td>float* for vmsFdim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdFdim</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdFdim</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The v\( ? \)Fdim function returns a vector containing the differences of the corresponding elements of the first and second vector arguments if the first element is larger, and +0 otherwise.

**Special values for Real Function v\( ? \)Fdim(\( x, y \))**

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>QNAN</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>any</td>
<td>SNAN</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
<tr>
<td>QNAN</td>
<td>any</td>
<td>QNAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNAN</td>
<td>any</td>
<td>QNAN</td>
<td></td>
<td>INVALID</td>
</tr>
</tbody>
</table>

**v\( ? \)Fmax**

*Returns the larger of each pair of elements of the two vector arguments.*

**Syntax**

vsFmax \( (n, a, b, y); \)

vmsFmax \( (n, a, b, y, mode); \)
vdFmax (n, a, b, y);
vmdFmax (n, a, b, y, mode);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float*</td>
<td>Pointers to the arrays containing the input vectors a and b.</td>
</tr>
<tr>
<td></td>
<td>for vsFmax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmsFmax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdFmax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vmdFmax</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsFmax</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsFmax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdFmax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdFmax</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Fmax function returns a vector with element values equal to the larger value from each pair of corresponding elements of the two vectors a and b: if \( a_i < b_i \), \( v?Fmax \) returns \( b_i \), otherwise \( v?Fmax \) returns \( a_i \).

Special values for Real Function v?Fmax(x, y)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>a; not NAN</td>
<td>NAN</td>
<td>a;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>b; not NAN</td>
<td>b;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>NAN</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Also

Fmin Returns the smaller of each pair of elements of the two vector arguments.
MaxMag Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

v?Fmin

Returns the smaller of each pair of elements of the two vector arguments.
Syntax

vsFmin (n, a, b, y);
vmsFmin (n, a, b, y, mode);
vdFmin (n, a, b, y);
vmdFmin (n, a, b, y, mode);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsFmin</td>
<td>Pointers to the arrays containing the input vectors a and b.</td>
</tr>
<tr>
<td></td>
<td>const float* for vmsFmin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vdFmin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vmdFmin</td>
<td></td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsFmin</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
<tr>
<td></td>
<td>float* for vmsFmin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vdFmin</td>
<td></td>
</tr>
<tr>
<td></td>
<td>double* for vmdFmin</td>
<td></td>
</tr>
</tbody>
</table>

Description

The v?Fmin function returns a vector with element values equal to the smaller value from each pair of corresponding elements of the two vectors a and b: if \( b_i < a_i \), v?Fmin returns \( b_i \), otherwise v?Fmin returns \( a_i \).

Special values for Real Function v?Fmin(x, y)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_i not NAN</td>
<td>NAN</td>
<td>a_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>b_i not NAN</td>
<td>b_i</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Also

Fmax Returns the larger of each pair of elements of the two vector arguments.
MinMag Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.
v?MaxMag

Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

Syntax

vsMaxMag (n, a, b, y);
vmsMaxMag (n, a, b, y, mode);
vMaxMag (n, a, b, y);
vmdMaxMag (n, a, b, y, mode);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>a, b</td>
<td>const float* for vsMaxMag const float* for vmsMaxMag const double* for vdMaxMag const double* for vmdMaxMag</td>
<td>Pointers to the arrays containing the input vectors a and b.</td>
</tr>
<tr>
<td>mode</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM mode setting for this function call. See vmlSetMode for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>float* for vsMaxMag float* for vmsMaxMag float* for vdMaxMag float* for vmdMaxMag</td>
<td>Pointer to an array containing the output vector y.</td>
</tr>
</tbody>
</table>

description

The v?MaxMag function returns a vector with element values equal to the element with the larger magnitude from each pair of corresponding elements of the two vectors a and b:

- If |a_i| > |b_i| v?MaxMag returns a_i, otherwise v?MaxMag returns a_i.
- If |b_i| > |a_i| v?MaxMag returns b_i, otherwise v?MaxMag returns a_i.
- Otherwise v?MaxMag behaves like v?Fmax.
Special values for Real Function \( v\text{?MaxMag}(x, y) \)

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>not NAN</td>
<td>NaN</td>
<td>( a_i )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>not NAN</td>
<td>( b_j )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Also

**MinMag** Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.

**Fmax** Returns the larger of each pair of elements of the two vector arguments.

**\( v\text{?MinMag} \)**

*Returns the element with the smaller magnitude between each pair of elements of the two vector arguments.*

Syntax

\[
\begin{align*}
\text{vsMinMag} & (n, a, b, y); \\
\text{vmsMinMag} & (n, a, b, y, mode); \\
\text{vdMinMag} & (n, a, b, y); \\
\text{vmdMinMag} & (n, a, b, y, mode);
\end{align*}
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Specifies the number of elements to be calculated.</td>
</tr>
<tr>
<td>( a, b )</td>
<td>const float* for vsMinMag const float* for vmsMinMag const double* for vdMinMag const double* for vmdMinMag</td>
<td>Pointers to the arrays containing the input vectors ( a ) and ( b ).</td>
</tr>
<tr>
<td>( mode )</td>
<td>const MKL_INT64</td>
<td>Overrides the global VM \textit{mode} setting for this function call. See \textit{vmlSetMode} for possible values and their description.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y )</td>
<td>float* for vsMinMag float* for vmsMinMag double* for vdMinMag</td>
<td>Pointer to an array containing the output vector ( y ).</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
 | double* | for vmdMinMag

**Description**

The `v?MinMag` function returns a vector with element values equal to the element with the smaller magnitude from each pair of corresponding elements of the two vectors `a` and `b`:

- If `|a_i| < |b_i|` `v?MaxMag` returns `a_i`, otherwise `v?MaxMag` returns `a_i`.
- If `|b_i| < |a_i|` `v?MaxMag` returns `b_i`, otherwise `v?MaxMag` returns `a_i`.
- Otherwise `v?MaxMag` behaves like `v?Fmin`.

**Special values for Real Function `v?MinMag(x, y)`**

<table>
<thead>
<tr>
<th>Argument 1</th>
<th>Argument 2</th>
<th>Result</th>
<th>VM Error Status</th>
<th>Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_i</code> not NAN</td>
<td>NAN</td>
<td><code>a_i</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td><code>b_i</code> not NAN</td>
<td><code>b_i</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>NAN</td>
<td>NAN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**See Also**

`MaxMag` Returns the element with the larger magnitude between each pair of elements of the two vector arguments.

`Fmin` Returns the smaller of each pair of elements of the two vector arguments.
Statistical Functions

Statistical functions in Intel® MKL are known as the Vector Statistics (VS). They are designed for the purpose of

- generating vectors of pseudorandom, quasi-random, and non-deterministic random numbers
- performing mathematical operations of convolution and correlation
- computing basic statistical estimates for single and double precision multi-dimensional datasets

The corresponding functionality is described in the respective Random Number Generators, Convolution and Correlation, and Summary Statistics sections.


The basic notion in VS is a task. The task object is a data structure or descriptor that holds the parameters related to a specific statistical operation: random number generation, convolution and correlation, or summary statistics estimation. Such parameters can be an identifier of a random number generator, its internal state and parameters, data arrays, their shape and dimensions, an identifier of the operation and so forth. You can modify the VS task parameters using the VS service functions.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

Random Number Generators

Intel® MKL VS provides a set of routines implementing commonly used pseudorandom, quasi-random, or non-deterministic random number generators with continuous and discrete distribution. To improve performance, all these routines were developed using the calls to the highly optimized Basic Random Number Generators (BRNGs) and vector mathematical functions (VM, see Chapter 9, "Vector Mathematical Functions").

VS provides interfaces both for Fortran and C languages. For users of the C and C++ languages the mkl_vsl.h header file is provided. All header files are found in the following directory:

{MKL}/include

All VS routines can be classified into three major categories:

- Transformation routines for different types of statistical distributions, for example, uniform, normal (Gaussian), binomial, etc. These routines indirectly call basic random number generators, which are pseudorandom, quasi-random, or non-deterministic random number generators. Detailed description of the generators can be found in Distribution Generators section.
- Service routines to handle random number streams: create, initialize, delete, copy, save to a binary file, load from a binary file, get the index of a basic generator. The description of these routines can be found in Service Routines section.
- Registration routines for basic pseudorandom generators and routines that obtain properties of the registered generators (see Advanced Service Routines section).
The last two categories are referred to as service routines.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Random Number Generators Conventions

This document makes no specific differentiation between random, pseudorandom, and quasi-random numbers, nor between random, pseudorandom, and quasi-random number generators unless the context requires otherwise. For details, refer to the 'Random Numbers' section in VS Notes document provided at the Intel® MKL web page.

All generators of nonuniform distributions, both discrete and continuous, are built on the basis of the uniform distribution generators, called Basic Random Number Generators (BRNGs). The pseudorandom numbers with nonuniform distribution are obtained through an appropriate transformation of the uniformly distributed pseudorandom numbers. Such transformations are referred to as generation methods. For a given distribution, several generation methods can be used. See VS Notes for the description of methods available for each generator.

An RNG task determines environment in which random number generation is performed, in particular parameters of the BRNG and its internal state. Output of VS generators is a stream of random numbers that are used in Monte Carlo simulations. A random stream descriptor and a random stream are used as synonyms of an RNG task in the document unless the context requires otherwise.

The random stream descriptor specifies which BRNG should be used in a given transformation method. See the Random Streams and RNGs in Parallel Computation section of VS Notes.

The term computational node means a logical or physical unit that can process data in parallel.

Random Number Generators Mathematical Notation

The following notation is used throughout the text:

\[ \mathbb{N} \quad \text{The set of natural numbers} \quad N = \{1, 2, 3 \ldots\}. \]

\[ \mathbb{Z} \quad \text{The set of integers} \quad Z = \{\ldots -3, -2, -1, 0, 1, 2, 3 \ldots\}. \]

\[ \mathbb{R} \quad \text{The set of real numbers}. \]

\[ \lfloor a \rfloor \quad \text{The floor of } a \text{ (the largest integer less than or equal to } a). \]

\[ \oplus \text{ or xor} \quad \text{Bitwise exclusive OR.} \]

\[ \binom{k}{\alpha} \quad \text{Binomial coefficient or combination} \quad (\alpha \in \mathbb{R}, \alpha \geq 0; k \in \mathbb{N} \cup \{0\}). \]

\[ C^0_\alpha = 1 \]

For \( \alpha \geq k \) binomial coefficient is defined as
If $\alpha < k$, then
\[ C^k_{\alpha} = 0 \]

$\Phi(x)$
Cumulative Gaussian distribution function
\[ \Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{y^2}{2} \right) dy \]
defined over $-\infty < x < +\infty$.
$\Phi(-\infty) = 0$, $\Phi(+\infty) = 1$.

$\Gamma(\alpha)$
The complete gamma function
\[ \Gamma(\alpha) = \int_{0}^{\infty} t^{\alpha-1} e^{-t} \, dt \]
where $\alpha > 0$.

$B(p, q)$
The complete beta function
\[ B(p, q) = \int_{0}^{1} t^{p-1}(1-t)^{q-1} \, dt \]
where $p>0$ and $q>0$.

$\text{LCG}(a, c, m)$
Linear Congruential Generator $x_{n+1} = (ax_n + c) \mod m$, where $a$ is called the multiplier, $c$ is called the increment, and $m$ is called the modulus of the generator.

$\text{MCG}(a, m)$
Multiplicative Congruential Generator $x_{n+1} = (ax_n) \mod m$ is a special case of Linear Congruential Generator, where the increment $c$ is taken to be 0.

$\text{GFSR}(p, q)$
Generalized Feedback Shift Register Generator
\[ x_n = x_{n-p} \oplus x_{n-q}. \]

Random Number Generators Naming Conventions

The names of the routines, types, and constants in VS random number generators are case-sensitive and can contain lowercase and uppercase characters ($\text{viRngUniform}$).

The names of generator routines have the following structure:
$v<\text{type of result}>\text{Rng}<\text{distribution}>$
where
- $v$ is the prefix of a VS vector function.
- $<\text{type of result}>$ is either $s$, $d$, or $i$ and specifies one of the following types:
Prefixes *s* and *d* apply to continuous distributions only, prefix *i* applies only to discrete case.

- *rng* indicates that the routine is a random generator.
- *<distribution>* specifies the type of statistical distribution.

Names of service routines follow the template below:

```
vs1<name>
```

where

- *vs1* is the prefix of a VS service function.
- *<name>* contains a short function name.

For a more detailed description of service routines, refer to Service Routines and Advanced Service Routines sections.

The prototype of each generator routine corresponding to a given probability distribution fits the following structure:

```
status = <function name>( method, stream, n, r, [<distribution parameters>] )
```

where

- *method* defines the method of generation. A detailed description of this parameter can be found in table "Values of *method* in method parameter". See the next page, where the structure of the *method* parameter name is explained.
- *stream* defines the descriptor of the random stream and must have a non-zero value. Random streams, descriptors, and their usage are discussed further in Random Streams and Service Routines.
- *n* defines the number of random values to be generated. If *n* is less than or equal to zero, no values are generated. Furthermore, if *n* is negative, an error condition is set.
- *r* defines the destination array for the generated numbers. The dimension of the array must be large enough to store at least *n* random numbers.
- *status* defines the error status of a VS routine. See Error Reporting section for a detailed description of error status values.

Additional parameters included into *<distribution parameters>* field are individual for each generator routine and are described in detail in Distribution Generators section.

To invoke a distribution generator, use a call to the respective VS routine. For example, to obtain a vector *r*, composed of *n* independent and identically distributed random numbers with normal (Gaussian) distribution, that have the mean value *a* and standard deviation *sigma*, write the following:

```
status = vsRngGaussian( method, stream, n, r, a, sigma )
```

The name of a *method* parameter has the following structure:

```
VSL_RNG_METHOD_<distribution>_VSL_RNG_METHOD_<method>_ACCURATE
```

where

- *<distribution>* is the probability distribution.
- *<method>* is the method name.

Type of the name structure for the *method* parameter corresponds to fast and accurate modes of random number generation (see "Distribution Generators" section and VS Notes for details).

Method names `VSL_RNG_METHOD_<distribution>_VSL_RNG_METHOD_<method>_ACCURATE` and
VSL_RNG_METHOD_<distribution>_<method>_ACCURATE

should be used with
v<precision>Rng<distribution>
function only, where

- `<precision>` is
  
  | s | for single precision continuous distribution |
  | d | for double precision continuous distribution |
  | i | for discrete distribution |

- `<distribution>` is the probability distribution.

is the probability distribution. Table "Values of `<method>` in method parameter" provides specific predefined values of the method name. The third column contains names of the functions that use the given method.

**Values of `<method>` in method parameter**

<table>
<thead>
<tr>
<th>Method</th>
<th>Short Description</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>STD</td>
<td>Standard method. Currently there is only one method for these functions.</td>
<td>Uniform (continuous), Uniform (discrete), UniformBits, UniformBits32, UniformBits64</td>
</tr>
<tr>
<td>BOXMULLER</td>
<td>BOXMULLER generates normally distributed random number $x$ thru the pair of uniformly distributed numbers $u_1$ and $u_2$ according to the formula: $x = \sqrt{-2 \ln u_1 \sin 2\pi u_2}$</td>
<td>Gaussian, GaussianMV</td>
</tr>
<tr>
<td>BOXMULLER2</td>
<td>BOXMULLER2 generates normally distributed random numbers $x_1$ and $x_2$ thru the pair of uniformly distributed numbers $u_1$ and $u_2$ according to the formulas: $x_1 = \sqrt{-2 \ln u_1 \sin 2\pi u_2}$, $x_2 = \sqrt{-2 \ln u_1 \cos 2\pi u_2}$</td>
<td>Gaussian, GaussianMV, Lognormal</td>
</tr>
<tr>
<td>ICDF</td>
<td>Inverse cumulative distribution function method.</td>
<td>Exponential, Laplace, Weibull, Cauchy, Rayleigh, Gumbel, Bernoulli, Geometric, Gaussian, Gaussian, GaussianMV, Lognormal</td>
</tr>
<tr>
<td>GNORM</td>
<td>For $\alpha &gt; 1$, a gamma distributed random number is generated as a cube of properly scaled normal random number; for $0.6 \leq \alpha &lt; 1$, a gamma distributed random number is generated using rejection from Weibull distribution; for $\alpha &lt; 0.6$, a gamma</td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Method</th>
<th>Short Description</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CJA</td>
<td>distributed random number is obtained using transformation of exponential power distribution; for $a = 1$, gamma distribution is reduced to exponential distribution. For $\min(p, q) &gt; 1$, Cheng method is used; for $\min(p, q) &lt; 1$, Johnk method is used, if $q + k \cdot p^2 + c \leq 0$ ($k = 0.852...$, $c = -0.956...$) otherwise, Atkinson switching algorithm is used; for $\max(p, q) &lt; 1$, method of Johnk is used; for $\min(p, q) &lt; 1, \max(p, q) &gt; 1$, Atkinson switching algorithm is used (CJA stands for the first letters of Cheng, Johnk, Atkinson); for $p = 1$ or $q = 1$, inverse cumulative distribution function method is used; for $p = 1$ and $q = 1$, beta distribution is reduced to uniform distribution.</td>
<td>Beta</td>
</tr>
<tr>
<td>BTPE</td>
<td>Acceptance/rejection method for $n_{trial} \cdot \min(p, 1 - p) \geq 30$ with decomposition into 4 regions: 2 parallelograms, triangle, left exponential tail, right exponential tail.</td>
<td>Binomial</td>
</tr>
<tr>
<td>H2PE</td>
<td>Acceptance/rejection method for large mode of distribution with decomposition into 3 regions: rectangular, left exponential tail, right exponential tail.</td>
<td>Hypergeometric</td>
</tr>
<tr>
<td>PTPE</td>
<td>Acceptance/rejection method for $\lambda \geq 27$ with decomposition into 4 regions: 2 parallelograms, triangle, left exponential tail, right exponential tail; otherwise, table lookup method is used.</td>
<td>Poisson</td>
</tr>
<tr>
<td>POISNORM</td>
<td>for $\lambda \geq 1$, method based on Poisson inverse CDF approximation by Gaussian inverse CDF; for $\lambda &lt; 1$, table lookup method is used.</td>
<td>Poisson, PoissonV</td>
</tr>
<tr>
<td>NBAR</td>
<td>Acceptance/rejection method for $\frac{(\alpha - 1) \cdot (1 - p)}{p} \geq 100$ with decomposition into 5 regions: rectangular, 2 trapezoid, left exponential tail, right exponential tail.</td>
<td>NegBinomial</td>
</tr>
</tbody>
</table>
### Method: CHI2GAMMA

Random number generator of chi-square distribution with \( \nu \) degrees of freedom. To generate any successive random number \( x \) of the chi-square distribution:

- If \( \nu \) is 1 or 3, a chi-square distributed random number is generated as a sum of squares of \( \nu \) independent normal random numbers with mean value \( a = 0 \) and standard deviation \( \sigma = 1 \).
- If \( \nu \) is even and \( 2 \leq \nu \leq 16 \), a chi-square distributed random number is generated using the formula:
  \[
  x = -2 \ln \left( \prod_{i=1}^{\nu/2} u_i \right)
  \]
  where \( u_i \) are successive random numbers uniformly distributed over the interval \((0, 1)\).
- If \( \nu \geq 17 \) or \( \nu \) is odd and \( 5 \leq \nu \leq 15 \), a chi-square distribution is reduced to a Gamma distribution with these parameters:
  - Shape \( \alpha = \nu / 2 \)
  - Offset \( a = 0 \)
  - Scale factor \( \beta = 2 \)

The random numbers of the Gamma distribution are generated using the `VSL_RNG_METHOD_GAMMA_GNORM` method.

### Statistical Functions

#### ChiSquare

**NOTE**

In this document, routines are often referred to by their base name (`Gaussian`) when this does not lead to ambiguity. In the routine reference, the full name (`vsrnggaussian`, `vsRngGaussian`) is always used in prototypes and code examples.

### Basic Generators

VS provides pseudorandom, quasi-random, and non-deterministic random number generators. This includes the following BRNGs, which differ in speed and other properties:

- The 31-bit multiplicative congruential pseudorandom number generator `MCG(1132489760, 2^{31} -1)` [L’Ecuyer99]
- The 32-bit generalized feedback shift register pseudorandom number generator `GFSR(250,103)` [Kirkpatrick81]
- The combined multiple recursive pseudorandom number generator `MRG32k3a` [L’Ecuyer99a]
- The 59-bit multiplicative congruential pseudorandom number generator `MCG(13^{13}, 2^{59})` from NAG Numerical Libraries [NAG]
- Wichmann-Hill pseudorandom number generator (a set of 273 basic generators) from NAG Numerical Libraries [NAG]
- Mersenne Twister pseudorandom number generator `MT19937` [Matsumoto98] with period length \( 2^{19937} -1 \) of the produced sequence
- Set of 6024 Mersenne Twister pseudorandom number generators `MT2203` [Matsumoto98, Matsumoto00]. Each of them generates a sequence of period length equal to \( 2^{2203} -1 \). Parameters of the generators provide mutual independence of the corresponding sequences.
- SIMD-oriented Fast Mersenne Twister pseudorandom number generator `SFMT19937` [Saito08] with a period length equal to \( 2^{19937} -1 \) of the produced sequence.
- Sobol quasi-random number generator [Sobol76], [Bratley88], which works in arbitrary dimension. For dimensions greater than 40 the user should supply initialization parameters (initial direction numbers and primitive polynomials or direction numbers) by using `vslNewStreamEx` function. See additional details on interface for registration of the parameters in the library in `VS Notes`.

2031
• Niederreiter quasi-random number generator [Bratley92], which works in arbitrary dimension. For dimensions greater than 318 the user should supply initialization parameters (irreducible polynomials or direction numbers) by using \texttt{vslNewStreamEx} function. See additional details on interface for registration of the parameters in the library in VS Notes.

• Non-deterministic random number generator (RDRAND-based generators only) [AVX], [IntelSWMan].

\textbf{NOTE}

You can use a non-deterministic random number generator only if the underlying hardware supports it. For instructions on how to detect if an Intel CPU supports a non-deterministic random number generator see, for example, Chapter 8: Post-32nm Processor Instructions in [AVX] or Chapter 4: RdRand Instruction Usage in [BMT].

\textbf{NOTE}

The time required by some non-deterministic sources to generate a random number is not constant, so you might have to make multiple requests before the next random number is available. VS limits the number of retries for requests to the non-deterministic source to 10. You can redefine the maximum number of retries during the initialization of the non-deterministic random number generator with the \texttt{vslNewStreamEx} function.

For more details on the non-deterministic source implementation for Intel CPUs please refer to Section 7.3.17, Volume 1, Random Number Generator Instruction in [IntelSWMan] and Section 4.2.2, RdRand Retry Loop in [BMT].

• Philox4x32-10 counter-based pseudorandom number generator with a period of $2^{128}$ [Philox4x32X10][Salmon11].

• ARS-5 counter-based pseudorandom number generator with a period of $2^{128}$, which uses instructions from the AES-NI set [ARS][Salmon11].


VS provides means of registration of such user-designed generators through the steps described in Advanced Service Routines section.

For some basic generators, VS provides two methods of creating independent random streams in multiprocessor computations, which are the leapfrog method and the block-splitting method. These sequence splitting methods are also useful in sequential Monte Carlo.

In addition, MT2023 pseudorandom number generator is a set of 6024 generators designed to create up to 6024 independent random sequences, which might be used in parallel Monte Carlo simulations. Another generator that has the same feature is Wichmann-Hill. It allows creating up to 273 independent random streams. The properties of the generators designed for parallel computations are discussed in detail in [Coddington94].

You may want to design and use your own basic generators. VS provides means of registration of such user-designed generators through the steps described in Advanced Service Routines section.

There is also an option to utilize externally generated random numbers in VS distribution generator routines. For this purpose VS provides three additional basic random number generators:

- for external random data packed in 32-bit integer array
- for external random data stored in double precision floating-point array; data is supposed to be uniformly distributed over $(a,b)$ interval
- for external random data stored in single precision floating-point array; data is supposed to be uniformly distributed over $(a,b)$ interval.

Such basic generators are called the abstract basic random number generators.

See VS Notes for a more detailed description of the generator properties.
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

BRNG Parameter Definition

Predefined values for the `brng` input parameter are as follows:

Values of `brng` parameter

<table>
<thead>
<tr>
<th>Value</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_BRNG_MCG31</td>
<td>A 31-bit multiplicative congruential generator.</td>
</tr>
<tr>
<td>VSL_BRNG_R250</td>
<td>A generalized feedback shift register generator.</td>
</tr>
<tr>
<td>VSL_BRNG_MRG32K3A</td>
<td>A combined multiple recursive generator with two components of order 3.</td>
</tr>
<tr>
<td>VSL_BRNG_MCG59</td>
<td>A 59-bit multiplicative congruential generator.</td>
</tr>
<tr>
<td>VSL_BRNG_WH</td>
<td>A set of 273 Wichmann-Hill combined multiplicative congruential generators.</td>
</tr>
<tr>
<td>VSL_BRNG_MT19937</td>
<td>A Mersenne Twister pseudorandom number generator.</td>
</tr>
<tr>
<td>VSL_BRNG_MT2203</td>
<td>A set of 6024 Mersenne Twister pseudorandom number generators.</td>
</tr>
<tr>
<td>VSL_BRNG_SFMT19937</td>
<td>A SIMD-oriented Fast Mersenne Twister pseudorandom number generator.</td>
</tr>
<tr>
<td>VSL_BRNG_SOBOL</td>
<td>A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 40$; user-defined dimensions are also available.</td>
</tr>
<tr>
<td>VSL_BRNG_NIEDERR</td>
<td>A 32-bit Gray code-based generator producing low-discrepancy sequences for dimensions $1 \leq s \leq 318$; user-defined dimensions are also available.</td>
</tr>
<tr>
<td>VSL_BRNG_IABSTRACT</td>
<td>An abstract random number generator for integer arrays.</td>
</tr>
<tr>
<td>VSL_BRNG_DABSTRACT</td>
<td>An abstract random number generator for double precision floating-point arrays.</td>
</tr>
<tr>
<td>VSL_BRNG_SABSTRACT</td>
<td>An abstract random number generator for single precision floating-point arrays.</td>
</tr>
<tr>
<td>VSL_BRNG_NONDETERM</td>
<td>A non-deterministic random number generator.</td>
</tr>
<tr>
<td>VSL_BRNG_PHILOX4X32X10</td>
<td>A Philox4x32-10 counter-based pseudorandom number generator.</td>
</tr>
</tbody>
</table>
**Value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_BRNG_ARS5</td>
<td>An ARS-5 counter-based pseudorandom number generator that uses instructions from the AES-NI set.</td>
</tr>
</tbody>
</table>

See VS Notes for detailed description.

### Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Random Streams**

A random stream (or stream) is an abstract source of pseudo- and quasi-random sequences of uniform distribution. You can operate with stream state descriptors only. A stream state descriptor, which holds state descriptive information for a particular BRNG, is a necessary parameter in each routine of a distribution generator. Only the distribution generator routines operate with random streams directly. See VS Notes for details.

**NOTE**

Random streams associated with abstract basic random number generator are called the abstract random streams. See VS Notes for detailed description of abstract streams and their use.

You can create unlimited number of random streams by VS Service Routines like NewStream and utilize them in any distribution generator to get the sequence of numbers of given probability distribution. When they are no longer needed, the streams should be deleted calling service routine DeleteStream.

VS provides service functions SaveStreamF and LoadStreamF to save random stream descriptive data to a binary file and to read this data from a binary file respectively. See VS Notes for detailed description.

**BRNG Data Types**

```c
typedef(void*) VSLStreamStatePtr;
```

See Advanced Service Routines for the format of the stream state structure for user-designed generators.

**Error Reporting**

VS RNG routines return status codes of the performed operation to report errors to the calling program. The application should perform error-related actions and/or recover from the error. The status codes are of integer type and have the following format:

- `VSL_ERROR_<ERROR_NAME>` - indicates VS errors common for all VS domains.
- `VSL_RNG_ERROR_<ERROR_NAME>` - indicates VS RNG errors.

VS RNG errors are of negative values while warnings are of positive values. The status code of zero value indicates successful completion of the operation: `VSL_ERROR_OK` (or synonymic `VSL_STATUS_OK`).
### Status Codes

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common VSL</strong></td>
<td></td>
</tr>
<tr>
<td>VSL_ERROR_OK, VSL_STATUS_OK</td>
<td>No error, execution is successful.</td>
</tr>
<tr>
<td>VSL_ERROR_BADARGS</td>
<td>Input argument value is not valid.</td>
</tr>
<tr>
<td>VSL_ERROR_CPU_NOT_SUPPORTED</td>
<td>CPU version is not supported.</td>
</tr>
<tr>
<td>VSL_ERROR.Feature_Not_Implemented</td>
<td>Feature invoked is not implemented.</td>
</tr>
<tr>
<td>VSL_ERROR_MEM_FAILURE</td>
<td>System cannot allocate memory.</td>
</tr>
<tr>
<td>VSL_ERROR_NULL_PTR</td>
<td>Input pointer argument is NULL.</td>
</tr>
<tr>
<td>VSL_ERROR_UNKNOWN</td>
<td>Unknown error.</td>
</tr>
<tr>
<td><strong>VS RNG Specific</strong></td>
<td></td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_FILE_FORMAT</td>
<td>File format is unknown.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_MEM_FORMAT</td>
<td>Descriptive random stream format is unknown.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_NBITS</td>
<td>The value in NBits field is bad.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_NSEEDS</td>
<td>The value in NSeeds field is bad.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_STREAM</td>
<td>The random stream is invalid.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE</td>
<td>The value in StreamStateSize field is bad.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_UPDATE</td>
<td>Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, &lt; 0 or &gt;nmax.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BAD_WORD_SIZE</td>
<td>The value in WordSize field is bad.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BRNG_NOT_SUPPORTED</td>
<td>BRNG is not supported by the function.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BRNG_TABLE_FULL</td>
<td>Registration cannot be completed due to lack of free entries in the table of registered BRNGs.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_BRNGS_INCOMPATIBLE</td>
<td>Two BRNGs are not compatible for the operation.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_FILE_CLOSE</td>
<td>Error in closing the file.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_FILE_OPEN</td>
<td>Error in opening the file.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_FILE_READ</td>
<td>Error in reading the file.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_FILE_WRITE</td>
<td>Error in writing the file.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_INVALID_ABSTRACT_STREAM</td>
<td>The abstract random stream is invalid.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_INVALID_BRNG_INDEX</td>
<td>BRNG index is not valid.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED</td>
<td>BRNG does not support Leapfrog method.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_NO_NUMBERS</td>
<td>Callback function for an abstract BRNG returns zero as the number of updated entries in a buffer.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED</td>
<td>Period of the generator is exceeded.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED</td>
<td>BRNG does not support Skip-Ahead method.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_SKIPAHEADEX_UNSUPPORTED</td>
<td>BRNG does not support advanced Skip-Ahead method.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_UNSUPPORTED_FILE_VER</td>
<td>File format version is not supported.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_NONDETERM_NOT_SUPPORTED</td>
<td>Non-deterministic random number generator is not supported on the CPU running the application.</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED</td>
<td>Number of retries to generate a random number using non-deterministic random number generator exceeds threshold (see Section 7.2.1.12 Non-deterministic in [VS Notes] for more details)</td>
</tr>
<tr>
<td>VSL_RNG_ERROR_AR5_NOT_SUPPORTED</td>
<td>ARS-5 random number generator is not supported on the CPU running the application.</td>
</tr>
</tbody>
</table>

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**VS RNG Usage Model RNG Usage Model**

A typical algorithm for VS random number generators is as follows:

2. Call one or more RNGs.
3. Process the output.
4. Delete the stream or streams with the function vslDeleteStream.

**NOTE**

You may reiterate steps 2-3. Random number streams may be generated for different threads.

The following example demonstrates generation of a random stream that is output of basic generator MT19937. The seed is equal to 777. The stream is used to generate 10,000 normally distributed random numbers in blocks of 1,000 random numbers with parameters $a = 5$ and $\sigma = 2$. Delete the streams after completing the generation. The purpose of the example is to calculate the sample mean for normal distribution with the given parameters.
Example of VS RNG Usage

```c
#include <stdio.h>
#include "mkl_vsl.h"

int main()
{
    double r[1000]; /* buffer for random numbers */
    double s; /* average */
    VSLStreamStatePtr stream;
    int i, j;

    /* Initializing */
    s = 0.0;
    vslNewStream( &stream, VSL_BRNG_MT19937, 777 );

    /* Generating */
    for ( i=0; i<10; i++ ) {
        vdRngGaussian( VSL_RNG_METHOD_GAUSSIAN_ICDF, stream, 1000, r, 5.0, 2.0 );
        for ( j=0; j<1000; j++ ) {
            s += r[j];
        }
    }
    s /= 10000.0;

    /* Deleting the stream */
    vslDeleteStream( &stream );

    /* Printing results */
    printf( "Sample mean of normal distribution = %f\n", s );
    return 0;
}
```

Additionally, examples that demonstrate usage of VS random number generators are available in:

```
${MKL}/examples/vslc/source
```

## Service Routines

Stream handling comprises routines for creating, deleting, or copying the streams and getting the index of a basic generator. A random stream can also be saved to and then read from a binary file. Table "Service Routines” lists all available service routines

### Service Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vslNewStream</td>
<td>Creates and initializes a random stream.</td>
</tr>
<tr>
<td>vslNewStreamEx</td>
<td>Creates and initializes a random stream for the generators with multiple initial conditions.</td>
</tr>
<tr>
<td>vsliNewAbstractStream</td>
<td>Creates and initializes an abstract random stream for integer arrays.</td>
</tr>
<tr>
<td>vsldNewAbstractStream</td>
<td>Creates and initializes an abstract random stream for double precision floating-point arrays.</td>
</tr>
<tr>
<td>vslsNewAbstractStream</td>
<td>Creates and initializes an abstract random stream for single precision floating-point arrays.</td>
</tr>
<tr>
<td>Routine</td>
<td>Short Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>vslDeleteStream</td>
<td>Deletes previously created stream.</td>
</tr>
<tr>
<td>vslCopyStream</td>
<td>Copies a stream to another stream.</td>
</tr>
<tr>
<td>vslCopyStreamState</td>
<td>Creates a copy of a random stream state.</td>
</tr>
<tr>
<td>vslSaveStreamF</td>
<td>Writes a stream to a binary file.</td>
</tr>
<tr>
<td>vslLoadStreamF</td>
<td>Reads a stream from a binary file.</td>
</tr>
<tr>
<td>vslSaveStreamM</td>
<td>Writes a random stream descriptive data, including state, to a memory buffer.</td>
</tr>
<tr>
<td>vslLoadStreamM</td>
<td>Creates a new stream and reads stream descriptive data, including state, from the memory buffer.</td>
</tr>
<tr>
<td>vslGetStreamSize</td>
<td>Computes size of memory necessary to hold the random stream.</td>
</tr>
<tr>
<td>vslLeapfrogStream</td>
<td>Initializes the stream by the leapfrog method to generate a subsequence of the original sequence.</td>
</tr>
<tr>
<td>vslSkipAheadStream</td>
<td>Initializes the stream by the skip-ahead method.</td>
</tr>
<tr>
<td>vslSkipAheadStreamEx</td>
<td>Initializes the stream by the advanced skip-ahead method.</td>
</tr>
<tr>
<td>vslGetStreamStateBrng</td>
<td>Obtains the index of the basic generator responsible for the generation of a given random stream.</td>
</tr>
<tr>
<td>vslGetNumRegBrngs</td>
<td>Obtains the number of currently registered basic generators.</td>
</tr>
</tbody>
</table>

Most of the generator-based work comprises three basic steps:

2. Generating random numbers with given distribution, see Distribution Generators.
3. Deleting the stream (vslDeleteStream).

Note that you can concurrently create multiple streams and obtain random data from one or several generators by using the stream state. You must use the vslDeleteStream function to delete all the streams afterwards.

**vslNewStream**

*Creates and initializes a random stream.*

**Syntax**

```c
status = vslNewStream( &stream, brng, seed );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brng</td>
<td>const MKL_INT</td>
<td>Index of the basic generator to initialize the stream. See Table Values of brng parameter for specific value.</td>
</tr>
</tbody>
</table>
**Name**  
*seed*  

**Type**  
const MKL_UINT  

**Description**  
Initial condition of the stream. In the case of a quasi-random number generator seed parameter is used to set the dimension. If the dimension is greater than the dimension that brng can support or is less than 1, then the dimension is assumed to be equal to 1.

**Output Parameters**

**Name**  
*stream*  

**Type**  
VSLStreamStatePtr*  

**Description**  
Stream state descriptor

**Description**

For a basic generator with number *brng*, this function creates a new stream and initializes it with a 32-bit seed. The seed is an initial value used to select a particular sequence generated by the basic generator *brng*. The function is also applicable for generators with multiple initial conditions. Use this function to create and initialize a new stream with a 32-bit seed only. If you need to provide multiple initial conditions such as several 32-bit or wider seeds, use the function vslNewStreamEx. See VS Notes for a more detailed description of stream initialization for different basic generators.

**NOTE**

This function is not applicable for abstract basic random number generators. Please use vsl1NewAbstractStream, vslsNewAbstractStream or vsldNewAbstractStream to utilize integer, single-precision or double-precision external random data respectively.

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- **VSL_ERROR_OK**, **VSL_STATUS_OK**  
  Indicates no error, execution is successful.
- **VSL_RNG_ERROR_INVALID_BRNG_INDEX**  
  BRNG index is invalid.
- **VSL_ERROR_MEM_FAILURE**  
  System cannot allocate memory for *stream*.
- **VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED**  
  Non-deterministic random number generator is not supported.
- **VSL_RNG_ERROR_ARS5_NOT_SUPPORTED**  
  ARS-5 random number generator is not supported on the CPU running the application.
vslNewStreamEx
Creates and initializes a random stream for generators with multiple initial conditions.

Syntax

```c
status = vslNewStreamEx( &stream, brng, n, params );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brng</td>
<td>const MKL_INT</td>
<td>Index of the basic generator to initialize the stream. See Table &quot;Values of brng parameter&quot; for specific value.</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of initial conditions contained in params</td>
</tr>
<tr>
<td>params</td>
<td>const unsigned int</td>
<td>Array of initial conditions necessary for the basic generator brng to initialize the stream. In the case of a quasi-random number generator only the first element in params parameter is used to set the dimension. If the dimension is greater than the dimension that brng can support or is less than 1, then the dimension is assumed to be equal to 1.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr*</td>
<td>Stream state descriptor</td>
</tr>
</tbody>
</table>

Description

The vslNewStreamEx function provides an advanced tool to set the initial conditions for a basic generator if its input arguments imply several initialization parameters. Initial values are used to select a particular sequence generated by the basic generator brng. Whenever possible, use vslNewStream, which is analogous to vslNewStreamEx except that it takes only one 32-bit initial condition. In particular, vslNewStreamEx may be used to initialize the state table in Generalized Feedback Shift Register Generators (GFSRs). A more detailed description of this issue can be found in VS Notes.

This function is also used to pass user-defined initialization parameters of quasi-random number generators into the library. See VS Notes for the format for their passing and registration in VS.

NOTE

This function is not applicable for abstract basic random number generators. Please use vsliNewAbstractStream, vslsNewAbstractStream or vsldNewAbstractStream to utilize integer, single-precision or double-precision external random data respectively.
## Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

## Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_RNG_ERROR_INVALID_BRNG_INDEX**: BRNG index is invalid.
- **VSL_ERROR_MEM_FAILURE**: System cannot allocate memory for stream.
- **VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED**: Non-deterministic random number generator is not supported.
- **VSL_RNG_ERROR_ARS5_NOT_SUPPORTED**: ARS-5 random number generator is not supported on the CPU running the application.

### vsl_iNewAbstractStream

*Creates and initializes an abstract random stream for integer arrays.*

#### Syntax

```
status = vsl_iNewAbstractStream( &stream, n, ibuf, icallback );
```

#### Include Files

- `mkl.h`

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
<td><code>const MKL_INT</code></td>
<td>Size of the array <code>ibuf</code></td>
</tr>
<tr>
<td><code>ibuf</code></td>
<td><code>const unsigned int</code></td>
<td>Array of <code>n</code> 32-bit integers</td>
</tr>
<tr>
<td><code>icallback</code></td>
<td></td>
<td>Pointer to the callback function used for <code>ibuf</code> update</td>
</tr>
</tbody>
</table>

#### NOTE

Format of the callback function:

```
int iUpdateFunc( VSLStreamStatePtr stream, int* n, unsigned int ibuf[], int* nmin, int* nmax, int* idx);
```

The callback function returns the number of elements in the array actually updated by the function. Table icallback Callback Function Parameters gives the description of the callback function parameters.
### icallback Callback Function Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>Abstract random stream descriptor</td>
</tr>
<tr>
<td>n</td>
<td>Size of ibuf</td>
</tr>
<tr>
<td>ibuf</td>
<td>Array of random numbers associated with the stream stream</td>
</tr>
<tr>
<td>nmin</td>
<td>Minimal quantity of numbers to update</td>
</tr>
<tr>
<td>nmax</td>
<td>Maximal quantity of numbers that can be updated</td>
</tr>
<tr>
<td>idx</td>
<td>Position in cyclic buffer ibuf to start update $0 \leq \text{idx} &lt; n$.</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr*</td>
<td>Descriptor of the stream state structure</td>
</tr>
</tbody>
</table>

### Description

The `vsliNewAbstractStream` function creates a new abstract stream and associates it with an integer array `ibuf` and your callback function `icallback` that is intended for updating of `ibuf` content.

### Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_BADARGS**: Parameter `n` is not positive.
- **VSL_ERROR_MEM_FAILURE**: System cannot allocate memory for `stream`.
- **VSL_ERROR_NULL_PTR**: Either buffer or callback function parameter is a NULL pointer.

### vsldNewAbstractStream

*Creates and initializes an abstract random stream for double precision floating-point arrays.*

### Syntax

```c
status = vsldNewAbstractStream( &stream, n,dbuf, a, b, dcallback );
```

### Include Files

- `mkl.h`

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Size of the array <code>dbuf</code></td>
</tr>
<tr>
<td>dbusf</td>
<td>const double</td>
<td>Array of <code>n</code> double precision floating-point random numbers with uniform distribution over interval <code>(a,b)</code></td>
</tr>
</tbody>
</table>
### Name | Type | Description
--- | --- | ---
*a* | const double | Left boundary *a*
*b* | const double | Right boundary *b*
*dcallback* | See Note below | Pointer to the callback function used for update of the array *dbuf*

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>stream</em></td>
<td>VSLStreamStatePtr*</td>
<td>Descriptor of the stream state structure</td>
</tr>
</tbody>
</table>

### NOTE
Format of the callback function:

```c
int dUpdateFunc(VSLStreamStatePtr stream, int* n, double dbuf[], int* nmin, int* nmax, int* idx);
```

The callback function returns the number of elements in the array actually updated by the function. Table *dcallback Callback Function Parameters* gives the description of the callback function parameters.

### *dcallback Callback Function Parameters*

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>stream</em></td>
<td>Abstract random stream descriptor</td>
</tr>
<tr>
<td><em>n</em></td>
<td>Size of <em>dbuf</em></td>
</tr>
<tr>
<td><em>dbuf</em></td>
<td>Array of random numbers associated with the stream <em>stream</em></td>
</tr>
<tr>
<td><em>nmin</em></td>
<td>Minimal quantity of numbers to update</td>
</tr>
<tr>
<td><em>nmax</em></td>
<td>Maximal quantity of numbers that can be updated</td>
</tr>
<tr>
<td><em>idx</em></td>
<td>Position in cyclic buffer <em>dbuf</em> to start update 0≤<em>idx</em>&lt;<em>n</em>.</td>
</tr>
</tbody>
</table>

### Description

The *vsldNewAbstractStream* function creates a new abstract stream for double precision floating-point arrays with random numbers of the uniform distribution over interval (*a*,*b*). The function associates the stream with a double precision array *dbuf* and your callback function *dcallback* that is intended for updating of *dbuf* content.

### Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK** Indicates no error, execution is successful.
- **VSL_ERROR_BADARGS** Parameter *n* is not positive.
- **VSL_ERROR_MEM_FAILURE** System cannot allocate memory for *stream*.
- **VSL_ERROR_NULL_PTR** Either buffer or callback function parameter is a NULL pointer.
**vslsNewAbstractStream**  
Creates and initializes an abstract random stream for single precision floating-point arrays.

**Syntax**

```c
status = vslsNewAbstractStream(&stream, n, sbuf, a, b, scallback);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Size of the array <code>sbuf</code></td>
</tr>
<tr>
<td><code>sbuf</code></td>
<td>const float</td>
<td>Array of <code>n</code> single precision floating-point random numbers with uniform distribution over interval <code>(a,b)</code></td>
</tr>
<tr>
<td><code>a</code></td>
<td>const float</td>
<td>Left boundary <code>a</code></td>
</tr>
<tr>
<td><code>b</code></td>
<td>const float</td>
<td>Right boundary <code>b</code></td>
</tr>
<tr>
<td><code>scallback</code></td>
<td>See <strong>Note</strong> below</td>
<td>Pointer to the callback function used for update of the array <code>sbuf</code></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>VSLStreamStatePtr*</td>
<td>Descriptor of the stream state structure</td>
</tr>
</tbody>
</table>

**NOTE**

Format of the callback function in C:

```c
int sUpdateFunc(VSLStreamStatePtr stream, int* n, float sbuf[], int* nmin, int* nmax, int* idx);
```

The callback function returns the number of elements in the array actually updated by the function. Table **scallback Callback Function Parameters** gives the description of the callback function parameters.

**scallback Callback Function Parameters**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>Abstract random stream descriptor</td>
</tr>
<tr>
<td><code>n</code></td>
<td>Size of <code>sbuf</code></td>
</tr>
<tr>
<td><code>sbuf</code></td>
<td>Array of random numbers associated with the stream <code>stream</code></td>
</tr>
<tr>
<td><code>nmin</code></td>
<td>Minimal quantity of numbers to update</td>
</tr>
<tr>
<td><code>nmax</code></td>
<td>Maximal quantity of numbers that can be updated</td>
</tr>
<tr>
<td><code>idx</code></td>
<td>Position in cyclic buffer <code>sbuf</code> to start update $0 \leq idx &lt; n$.</td>
</tr>
</tbody>
</table>
**Description**

The `vslNewAbstractStream` function creates a new abstract stream for single precision floating-point arrays with random numbers of the uniform distribution over interval $(a,b)$. The function associates the stream with a single precision array `sbuf` and your callback function `scallback` that is intended for updating of `sbuf` content.

**Return Values**

- `VSL_ERROR_OK, VSL_STATUS_OK` Indicates no error, execution is successful.
- `VSL_ERROR_BADARGS` Parameter `n` is not positive.
- `VSL_ERROR_MEM_FAILURE` System cannot allocate memory for `stream`.
- `VSL_ERROR_NULL_PTR` Either buffer or callback function parameter is a NULL pointer.

**vslDeleteStream**

*Deletes a random stream.*

**Syntax**

```c
status = vslDeleteStream( &stream );
```

**Include Files**

- `mkl.h`

**Input/Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td>VSLStreamStatePtr*</td>
<td>Stream state descriptor. Must have non-zero value. After the stream is successfully deleted, the pointer is set to NULL.</td>
</tr>
</tbody>
</table>

**Description**

The function deletes the random stream created by one of the initialization functions.

**Return Values**

- `VSL_ERROR_OK, VSL_STATUS_OK` Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR` `stream` parameter is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM` `stream` is not a valid random stream.

**vslCopyStream**

*Creates a copy of a random stream.*

**Syntax**

```c
status = vslCopyStream( &newstream, srcstream );
```
Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcstream</td>
<td>const VSLStreamStatePtr</td>
<td>Pointer to the stream state structure to be copied</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newstream</td>
<td>VSLStreamStatePtr*</td>
<td>Copied random stream descriptor</td>
</tr>
</tbody>
</table>

Description

The function creates an exact copy of `srcstream` and stores its descriptor to `newstream`.

Return Values

- VSL_ERROR_OK, VSL_STATUS_OK: Indicates no error, execution is successful.
- VSL_ERROR_NULL_PTR: `srcstream` parameter is a NULL pointer.
- VSL_RNG_ERROR_BAD_STREAM: `srcstream` is not a valid random stream.
- VSL_ERROR_MEM_FAILURE: System cannot allocate memory for `newstream`.

**vslCopyStreamState**

Creates a copy of a random stream state.

Syntax

```c
status = vslCopyStreamState( deststream, srcstream );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srcstream</td>
<td>const VSLStreamStatePtr</td>
<td>Pointer to the stream state structure, from which the state structure is copied</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deststream</td>
<td>VSLStreamStatePtr*</td>
<td>Pointer to the stream state structure where the stream state is copied</td>
</tr>
</tbody>
</table>
Description

The `vslCopyStreamState` function copies a stream state from `srcstream` to the existing `deststream` stream. Both the streams should be generated by the same basic generator. An error message is generated when the index of the BRNG that produced `deststream` stream differs from the index of the BRNG that generated `srcstream` stream.

Unlike `vslCopyStream` function, which creates a new stream and copies both the stream state and other data from `srcstream`, the function `vslCopyStreamState` copies only `srcstream` stream state data to the generated `deststream` stream.

Return Values

- `VSL_ERROR_OK`, `VSL_STATUS_OK`: Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`: Either `srcstream` or `deststream` is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`: Either `srcstream` or `deststream` is not a valid random stream.
- `VSL_RNG_ERROR_BRNGS_INCOMPATIBLE`: BRNG associated with `srcstream` is not compatible with BRNG associated with `deststream`.

`vslSaveStreamF`

Writes random stream descriptive data, including stream state, to binary file.

Syntax

```c
errstatus = vslSaveStreamF( stream, fname );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>const VSLStreamStatePtr</td>
<td>Random stream to be written to the file</td>
</tr>
<tr>
<td>fname</td>
<td>const char*</td>
<td>File name specified as a null-terminated string</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errstatus</td>
<td>int</td>
<td>Error status of the operation</td>
</tr>
</tbody>
</table>

Description

The `vslSaveStreamF` function writes the random stream descriptive data, including the stream state, to the binary file. Random stream descriptive data is saved to the binary file with the name `fname`. The random stream `stream` must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. If the stream cannot be saved to the file, `errstatus` has a non-zero value. The random stream can be read from the binary file using the `vslLoadStreamF` function.
Return Values

VSL_ERROR_OK, VSL_STATUS_OK  Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR          Either fname or stream is a NULL pointer.
VSL_RNG_ERROR_BAD_STREAM    stream is not a valid random stream.
VSL_RNG_ERROR_FILE_OPEN     Indicates an error in opening the file.
VSL_RNG_ERROR_FILE_WRITE    Indicates an error in writing the file.
VSL_RNG_ERROR_FILE_CLOSE    Indicates an error in closing the file.
VSL_ERROR_MEM_FAILURE      System cannot allocate memory for internal needs.

vslLoadStreamF

Creates new stream and reads stream descriptive
data, including stream state, from binary file.

Syntax

errstatus = vslLoadStreamF( &stream, fname );

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fname</td>
<td>const char*</td>
<td>File name specified as a null-terminated string</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr*</td>
<td>Pointer to a new random stream</td>
</tr>
<tr>
<td>errstatus</td>
<td>int</td>
<td>Error status of the operation</td>
</tr>
</tbody>
</table>

Description

The vslLoadStreamF function creates a new stream and reads stream descriptive data, including the stream state, from the binary file. A new random stream is created using the stream descriptive data from the binary file with the name fname. If the stream cannot be read (for example, an I/O error occurs or the file format is invalid), errstatus has a non-zero value. To save random stream to the file, use vslSaveStreamF function.

Caution

Calling vslLoadStreamF with a previously initialized stream pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling vslLoadStreamF, you should call the vslDeleteStream function first to deallocate the resources.
### Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**
  
  Indicates no error, execution is successful.

- **VSL_ERROR_NULL_PTR**
  
  *fname* is a NULL pointer.

- **VSL_RNG_ERROR_FILE_OPEN**
  
  Indicates an error in opening the file.

- **VSL_RNG_ERROR_FILE_WRITE**
  
  Indicates an error in writing the file.

- **VSL_RNG_ERROR_FILE_CLOSE**
  
  Indicates an error in closing the file.

- **VSL_ERROR_MEM_FAILURE**
  
  System cannot allocate memory for internal needs.

- **VSL_RNG_ERROR_BAD_FILE_FORMAT**
  
  Unknown file format.

- **VSL_RNG_ERROR_UNSUPPORTED_FILE_VER**
  
  File format version is unsupported.

- **VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED**
  
  Non-deterministic random number generator is not supported.

- **VSL_RNG_ERROR_AR5_NOT_SUPPORTED**
  
  ARS-5 random number generator is not supported on the CPU running the application.

### vslSaveStreamM

*Writes random stream descriptive data, including stream state, to a memory buffer.*

#### Syntax

```c
errstatus = vslSaveStreamM( stream, memptr );
```

#### Include Files

- mkl.h

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>const VSLStreamStatePtr</td>
<td>Random stream to be written to the memory</td>
</tr>
<tr>
<td>memptr</td>
<td>char*</td>
<td>Memory buffer to save random stream descriptive data to</td>
</tr>
</tbody>
</table>
### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>errstatus</td>
<td>int</td>
<td>Error status of the operation</td>
</tr>
</tbody>
</table>

### Description

The `vslSaveStreamM` function writes the random stream descriptive data, including the stream state, to the memory at `memptr`. Random stream `stream` must be a valid stream created by `vslNewStream`-like or `vslCopyStream`-like service routines. The `memptr` parameter must be a valid pointer to the memory of size sufficient to hold the random stream `stream`. Use the service routine `vslGetStreamSize` to determine this amount of memory.

If the stream cannot be saved to the memory, `errstatus` has a non-zero value. The random stream can be read from the memory pointed by `memptr` using the `vslLoadStreamM` function.

### Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**: Either `memptr` or `stream` is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**: `stream` is a NULL pointer.

### vslLoadStreamM

*Creates a new stream and reads stream descriptive data, including stream state, from the memory buffer.*

#### Syntax

```c
errstatus = vslLoadStreamM( &stream, memptr );
```

### Include Files

- `mkl.h`

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memptr</td>
<td>const char*</td>
<td>Memory buffer to load random stream descriptive data from</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr*</td>
<td>Pointer to a new random stream</td>
</tr>
<tr>
<td>errstatus</td>
<td>int</td>
<td>Error status of the operation</td>
</tr>
</tbody>
</table>

### Description

The `vslLoadStreamM` function creates a new stream and reads stream descriptive data, including the stream state, from the memory buffer. A new random stream is created using the stream descriptive data from the memory pointer by `memptr`. If the stream cannot be read (for example, `memptr` is invalid), `errstatus` has a non-zero value. To save random stream to the memory, use `vslSaveStreamM` function. Use the service routine `vslGetStreamSize` to determine the amount of memory sufficient to hold the random stream.
Caution
Calling `LoadStreamM` with a previously initialized `stream` pointer can have unintended consequences such as a memory leak. To initialize a stream which has been in use until calling `vslLoadStreamM`, you should call the `vslDeleteStream` function first to deallocate the resources.

Optimization Notice
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**
  Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**
  `memptr` is a NULL pointer.
- **VSL_ERROR_MEM_FAILURE**
  System cannot allocate memory for internal needs.
- **VSL_RNG_ERROR_BAD_MEM_FORMAT**
  Descriptive random stream format is unknown.
- **VSL_RNG_ERROR_NONDETERMINISTIC_NOT_SUPPORTED**
  Non-deterministic random number generator is not supported.
- **VSL_RNG_ERROR_AR5_NOT_SUPPORTED**
  ARS-5 random number generator is not supported on the CPU running the application.

`vslGetStreamSize`
*Computes size of memory necessary to hold the random stream.*

**Syntax**

```c
memsize = vslGetStreamSize(stream);
```

**Include Files**
- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>stream</code></td>
<td><code>const VSLStreamStatePtr</code></td>
<td>Random stream</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memsize</td>
<td>int</td>
<td>Amount of memory in bytes necessary to hold descriptive data of random stream</td>
</tr>
</tbody>
</table>

Description

The vslGetStreamSize function returns the size of memory in bytes which is necessary to hold the given random stream. Use the output of the function to allocate the buffer to which you will save the random stream by means of the vslSaveStreamM function.

Return Values

- VSL_ERROR_OK, VSL_STATUS_OK: Indicates no error, execution is successful.
- VSL_RNG_ERROR_BAD_STREAM: stream is a NULL pointer.

vslLeapfrogStream

*Initializes a stream using the leapfrog method.*

Syntax

```c
status = vslLeapfrogStream( stream, k, nstreams );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure to which leapfrog method is applied</td>
</tr>
<tr>
<td>k</td>
<td>const MKL_INT</td>
<td>Index of the computational node, or stream number</td>
</tr>
<tr>
<td>nstreams</td>
<td>const MKL_INT</td>
<td>Largest number of computational nodes, or stride</td>
</tr>
</tbody>
</table>

Description

The vslLeapfrogStream function generates random numbers in a random stream with non-unit stride. This feature is particularly useful in distributing random numbers from the original stream across the nstreams buffers without generating the original random sequence with subsequent manual distribution.

One of the important applications of the leapfrog method is splitting the original sequence into non-overlapping subsequences across nstreams computational nodes. The function initializes the original random stream (see Figure "Leapfrog Method") to generate random numbers for the computational node $k$, $0 \leq k < nstreams$, where nstreams is the largest number of computational nodes used.
Leapfrog Method

The leapfrog method is supported only for those basic generators that allow splitting elements by the leapfrog method, which is more efficient than simply generating them by a generator with subsequent manual distribution across computational nodes. See VS Notes for details.

For quasi-random basic generators, the leapfrog method allows generating individual components of quasi-random vectors instead of whole quasi-random vectors. In this case \( n_{streams} \) parameter should be equal to the dimension of the quasi-random vector while \( k \) parameter should be the index of a component to be generated \((0 \leq k < n_{streams})\). Other parameters values are not allowed.

The following code illustrates the initialization of three independent streams using the leapfrog method:

**Code for Leapfrog Method**

```c
VSLStreamStatePtr stream1;
VSLStreamStatePtr stream2;
VSLStreamStatePtr stream3;

/* Creating 3 identical streams */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
status = vslCopyStream(&stream2, stream1);
status = vslCopyStream(&stream3, stream1);

/* Leapfrogging the streams */
status = vslLeapfrogStream(stream1, 0, 3);
status = vslLeapfrogStream(stream2, 1, 3);
status = vslLeapfrogStream(stream3, 2, 3);

/* Generating random numbers */
...

/* Deleting the streams */
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...
```
**Return Values**

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**: Stream is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**: Stream is not a valid random stream.
- **VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED**: BRNG does not support Leapfrog method.

**vslSkipAheadStream**

*Initializes a stream using the block-splitting method.*

**Syntax**

```c
status = vslSkipAheadStream( stream, nskip);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure to which block-splitting method is applied</td>
</tr>
<tr>
<td>nskip</td>
<td>const long long int</td>
<td>Number of skipped elements</td>
</tr>
</tbody>
</table>

**Description**

The `vslSkipAheadStream` function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is `nskip`, then the original random sequence may be split by `vslSkipAheadStream` into non-overlapping blocks of `nskip` size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. (see Figure "Block-Splitting Method").
The skip-ahead method is supported only for those basic generators that allow skipping elements by the skip-ahead method, which is more efficient than simply generating them by generator with subsequent manual skipping. See VS Notes for details.

Please note that for quasi-random basic generators the skip-ahead method works with components of quasi-random vectors rather than with whole quasi-random vectors. Therefore, to skip \(N_S\) quasi-random vectors, set the \(n_{skip}\) parameter equal to the \(N_S \times \text{DIMEN}\), where \(\text{DIMEN}\) is the dimension of the quasi-random vector. If this operation results in exceeding the period of the quasi-random number generator, which is \(2^{32}-1\), the library returns the \(\text{VSL\_RNG\_ERROR\_QRNG\_PERIOD\_ELAPSED}\) error code.

The following code illustrates how to initialize three independent streams using the `vslSkipAheadStream` function:

**Code for Block-Splitting Method**

```c
VSLSmStreamStatePtr stream1;
VSLSmStreamStatePtr stream2;
VSLSmStreamStatePtr stream3;

/* Creating the 1st stream */
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);

/* Skipping ahead by 7 elements the 2nd stream */
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStream(stream2, 7);

/* Skipping ahead by 7 elements the 3rd stream */
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStream(stream3, 7);

/* Generating random numbers */
...

/* Deleting the streams */
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
...```

At node 1 the stream contains 1, 2, 3, 4, 5, 6, 7.
At node 2 the stream contains 8, 9, 10, 11, 12, 13, 14.
At node 3 the stream contains 15, 16, 17, 18, 19, 20, 21.
Return Values

- **VSL_ERROR_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**: Stream is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**: Stream is not a valid random stream.
- **VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED**: BRNG does not support the Skip-Ahead method.
- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**: Period of the quasi-random number generator is exceeded.

vslSkipAheadStreamEx

*Initializes a stream using the block-splitting method with partitioned number of skipped elements.*

Syntax

```c
status = vslSkipAheadStreamEx(stream, n, nskip);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure to which block-splitting method is applied</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of summands in <code>nskip</code></td>
</tr>
<tr>
<td>nskip</td>
<td>const MKL_UINT64[]</td>
<td>Partitioned number of skipped elements</td>
</tr>
</tbody>
</table>

Description

The vslSkipAheadStreamEx function skips a given number of elements in a random stream. This feature is particularly useful in distributing random numbers from original random stream across different computational nodes. If the largest number of random numbers used by a computational node is `nskip`, then the original random sequence may be split by vslSkipAheadStreamEx into non-overlapping blocks of `nskip` size so that each block corresponds to the respective computational node. The number of computational nodes is unlimited. This method is known as the block-splitting method or as the skip-ahead method. See Notes for Intel® MKL Vector Statistics for more details.

Use this function when the number of elements to skip in a random stream is greater than $2^{63}$. Prior calls to the function represent the number of skipped elements with array of size `n` as shown below:

$\text{nskip}[0] + \text{nskip}[1] \times 2^{64} + \text{nskip}[2] \times 2^{128} + \ldots + \text{nskip}[n-1] \times 2^{64 \times (n-1)}$;

When the number of skipped elements is less than $2^{63}$ you can use either vslSkipAheadStreamEx or vslSkipAheadStream. The following code illustrates how to initialize three independent streams using the vslSkipAheadStreamEx function:

```c
VSLStreamStatePtr stream1; VSLStreamStatePtr stream2; VSLStreamStatePtr stream3;

// Creating the 1st stream
status = vslNewStream(&stream1, VSL_BRNG_MCG31, 174);
```
To skip $2^{64}$ elements in the random stream, `SkipAheadStreamEx(nskip)` function should be called with `nskip` represented as $nskip = 2^{64} = 0 + 1 * 2^{64}$

```c
MKL_UINT64 nskip[2];
nskip[0]=0;
nskip[1]=1;
```

/* Skipping ahead by $2^{64}$ elements the 2nd stream */
```c
status = vslCopyStream(&stream2, stream1);
status = vslSkipAheadStreamEx (stream2, 2, nskip);
```

/* Skipping ahead by $2^{64}$ elements the 3rd stream */
```c
status = vslCopyStream(&stream3, stream2);
status = vslSkipAheadStreamEx (stream3, 2, nskip);
```

/* Generating random numbers */
```c
... */
```

/* Deleting the streams */
```c
status = vslDeleteStream(&stream1);
status = vslDeleteStream(&stream2);
status = vslDeleteStream(&stream3);
```

### Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**
  Indicating no error, execution is successful.
- **VSL_ERROR_NULL_PTR**
  `stream` is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**
  `stream` is not a valid random stream.
- **VSL_RNG_ERROR_SKIPAHEADEX_UNSUPPORTED**
  BRNG does not support the advanced Skip-Ahead method.

### vslGetStreamStateBrng

Returns index of a basic generator used for generation of a given random stream.

**Syntax**

```c
brng = vslGetStreamStateBrng( stream );
```

### Include Files

- `mkl.h`

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stream</td>
<td>const VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brng</td>
<td>int</td>
<td>Index of the basic generator assigned for the generation of stream; negative in case of an error</td>
</tr>
</tbody>
</table>

Description

The `vslGetStreamStateBrng` function retrieves the index of a basic generator used for generation of a given random stream.

Return Values

- `VSL_ERROR_NULL_PTR`: stream is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`: stream is not a valid random stream.

`vslGetNumRegBrngs`

*Obtains the number of currently registered basic generators.*

Syntax

```c
nregbrngs = vslGetNumRegBrngs( void );
```

Include Files

- `mkl.h`

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nregbrngs</td>
<td>int</td>
<td>Number of basic generators registered at the moment of the function call</td>
</tr>
</tbody>
</table>

Description

The `vslGetNumRegBrngs` function obtains the number of currently registered basic generators. Whenever user registers a user-designed basic generator, the number of registered basic generators is incremented. The maximum number of basic generators that can be registered is determined by the `VSL_MAX_REG_BRNGS` parameter.

Distribution Generators

VS routines are used to generate random numbers with different types of distribution. Each function group is introduced below by the type of underlying distribution and contains a short description of its functionality, as well as specifications of the call sequence and the explanation of input and output parameters. Table "Continuous Distribution Generators" and Table "Discrete Distribution Generators" list the random number generator routines with data types and output distributions, and sets correspondence between data types of the generator routines and the basic random number generators.
### Continuous Distribution Generators

<table>
<thead>
<tr>
<th>Type of Distribution</th>
<th>Data Types</th>
<th>BRNG Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRngUniform</td>
<td>s, d</td>
<td>s, d</td>
<td>Uniform continuous distribution on the interval ([a,b])</td>
</tr>
<tr>
<td>vRngGaussian</td>
<td>s, d</td>
<td>s, d</td>
<td>Normal (Gaussian) distribution</td>
</tr>
<tr>
<td>vRngGaussianMV</td>
<td>s, d</td>
<td>s, d</td>
<td>Multivariate normal (Gaussian) distribution</td>
</tr>
<tr>
<td>vRngExponential</td>
<td>s, d</td>
<td>s, d</td>
<td>Exponential distribution</td>
</tr>
<tr>
<td>vRngLaplace</td>
<td>s, d</td>
<td>s, d</td>
<td>Laplace distribution (double exponential distribution)</td>
</tr>
<tr>
<td>vRngWeibull</td>
<td>s, d</td>
<td>s, d</td>
<td>Weibull distribution</td>
</tr>
<tr>
<td>vRngCauchy</td>
<td>s, d</td>
<td>s, d</td>
<td>Cauchy distribution</td>
</tr>
<tr>
<td>vRngRayleigh</td>
<td>s, d</td>
<td>s, d</td>
<td>Rayleigh distribution</td>
</tr>
<tr>
<td>vRngLognormal</td>
<td>s, d</td>
<td>s, d</td>
<td>Lognormal distribution</td>
</tr>
<tr>
<td>vRngGumbel</td>
<td>s, d</td>
<td>s, d</td>
<td>Gumbel (extreme value) distribution</td>
</tr>
<tr>
<td>vRngGamma</td>
<td>s, d</td>
<td>s, d</td>
<td>Gamma distribution</td>
</tr>
<tr>
<td>vRngBeta</td>
<td>s, d</td>
<td>s, d</td>
<td>Beta distribution</td>
</tr>
<tr>
<td>vRngChiSquare</td>
<td>s, d</td>
<td>s, d</td>
<td>Chi-Square distribution</td>
</tr>
</tbody>
</table>

### Discrete Distribution Generators

<table>
<thead>
<tr>
<th>Type of Distribution</th>
<th>Data Types</th>
<th>BRNG Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRngUniform</td>
<td>i</td>
<td>d</td>
<td>Uniform discrete distribution on the interval ([a,b])</td>
</tr>
<tr>
<td>vRngUniformBits</td>
<td>i</td>
<td>i</td>
<td>Underlying BRNG integer recurrence</td>
</tr>
<tr>
<td>vRngUniformBits32</td>
<td>i</td>
<td>i</td>
<td>Uniformly distributed bits in 32-bit chunks</td>
</tr>
<tr>
<td>vRngUniformBits64</td>
<td>i</td>
<td>i</td>
<td>Uniformly distributed bits in 64-bit chunks</td>
</tr>
<tr>
<td>vRngBernoulli</td>
<td>i</td>
<td>s</td>
<td>Bernoulli distribution</td>
</tr>
<tr>
<td>vRngGeometric</td>
<td>i</td>
<td>s</td>
<td>Geometric distribution</td>
</tr>
<tr>
<td>vRngBinomial</td>
<td>i</td>
<td>d</td>
<td>Binomial distribution</td>
</tr>
<tr>
<td>vRngHypergeometric</td>
<td>i</td>
<td>d</td>
<td>Hypergeometric distribution</td>
</tr>
<tr>
<td>vRngPoisson</td>
<td>i</td>
<td>s</td>
<td>Poisson distribution (for (VSL_RNG_METHOD_POISSON_POISNORM)</td>
</tr>
<tr>
<td>Type of Distribution</td>
<td>BRNG Data Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>vRngUniform</td>
<td>s, d</td>
<td>Poisson distribution with varying mean</td>
<td></td>
</tr>
<tr>
<td>vRngExponential</td>
<td>s, d</td>
<td>Negative binomial distribution, or Pascal distribution</td>
<td></td>
</tr>
<tr>
<td>vRngWeibull</td>
<td>s, d</td>
<td>Multinomial distribution</td>
<td></td>
</tr>
</tbody>
</table>

**Modes of random number generation**

The library provides two modes of random number generation, accurate and fast. Accurate generation mode is intended for the applications that are highly demanding to accuracy of calculations. When used in this mode, the generators produce random numbers lying completely within definitional domain for all values of the distribution parameters. For example, random numbers obtained from the generator of continuous distribution that is uniform on interval \([a, b]\) belong to this interval irrespective of what \(a\) and \(b\) values may be. Fast mode provides high performance of generation and also guarantees that generated random numbers belong to the definitional domain except for some specific values of distribution parameters. The generation mode is set by specifying relevant value of the method parameter in generator routines. List of distributions that support accurate mode of generation is given in the table below.

**Distribution Generators Supporting Accurate Mode**

<table>
<thead>
<tr>
<th>Type of Distribution</th>
<th>Data Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRngUniform</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngExponential</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngWeibull</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngRayleigh</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngLognormal</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngGamma</td>
<td>s, d</td>
</tr>
<tr>
<td>vRngBeta</td>
<td>s, d</td>
</tr>
</tbody>
</table>

See additional details about accurate and fast mode of random number generation in VS Notes.

**New method names**

The current version of has a modified structure of VS RNG method names. (See RNG Naming Conventions for details.) The old names are kept for backward compatibility. The set correspondence between the new and legacy method names for VS random number generators.
### Method Names for Continuous Distribution Generators

<table>
<thead>
<tr>
<th>RNG</th>
<th>Legacy Method Name</th>
<th>New Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRngUniform</td>
<td>VSL_METHOD_SUNIFORM_STD,</td>
<td>VSL_RNG_METHOD_UNIFORM_STD,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DUNIFORM_STD,</td>
<td>VSL_RNG_METHOD_UNIFORM_STD_ACURATE</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SUNIFORM_STD_ACCURATE,</td>
<td>VSL_RNG_METHOD_UNIFORM_STD_ACCURATE</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DUNIFORM_STD_ACCURATE,</td>
<td>VSL_RNG_METHOD_UNIFORM_STD_ACCURATE</td>
</tr>
<tr>
<td>vRngGaussian</td>
<td>VSL_METHOD_SGAUSSIAN_BOXMULLER,</td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SGAUSSIAN_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SGAUSSIAN_ICDF,</td>
<td>VSL_RNG_METHOD_GAUSSIAN_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIAN_BOXMULLER,</td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIAN_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIAN_ICDF</td>
<td>VSL_RNG_METHOD_GAUSSIAN_ICDF</td>
</tr>
<tr>
<td>vRngGaussianMV</td>
<td>VSL_METHOD_SGAUSSIANMV_BOXMULLER,</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SGAUSSIANMV_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SGAUSSIANMV_ICDF</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIANMV_BOXMULLER,</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIANMV_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DGAUSSIANMV_ICDF</td>
<td>VSL_RNG_METHOD_GAUSSIANMV_ICDF</td>
</tr>
<tr>
<td>vRngExponential</td>
<td>VSL_METHOD_SEXPONENTIAL_ICDF,</td>
<td>VSL_RNG_METHOD_EXPONENTIAL_ICDF,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DEXPONENTIAL_ICDF,</td>
<td>VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SEXPONENTIAL_ICDF_ACCURATE,</td>
<td>VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DEXPONENTIAL_ICDF_ACCURATE</td>
<td>VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE,</td>
</tr>
<tr>
<td>vRngLaplace</td>
<td>VSL_METHOD_SLAPLACE_ICDF,</td>
<td>VSL_RNG_METHOD_LAPLACE_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DLAPLACEL_ICDF</td>
<td>VSL_RNG_METHOD_LAPLACE_ICDF</td>
</tr>
<tr>
<td>vRngWeibull</td>
<td>VSL_METHOD_SWEIBULL_ICDF,</td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DWEIBULL_ICDF,</td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SWEIBULL_ICDF_ACCURATE,</td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DWEIBULL_ICDF_ACCURATE,</td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF</td>
</tr>
<tr>
<td>vRngCauchy</td>
<td>VSL_METHOD_SCAUCHY_ICDF,</td>
<td>VSL_RNG_METHOD_CAUCHY_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DCAUCHY_ICDF</td>
<td>VSL_RNG_METHOD_CAUCHY_ICDF</td>
</tr>
<tr>
<td>vRngRayleigh</td>
<td>VSL_METHOD_SRAYLEIGH_ICDF,</td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DRAYLEIGH_ICDF</td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SRAYLEIGH_ICDF_ACCURATE,</td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DRAYLEIGH_ICDF_ACCURATE</td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF</td>
</tr>
<tr>
<td>vRngLognormal</td>
<td>VSL_METHOD_SLOGNORMAL_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DLOGNORMAL_BOXMULLER2,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SLOGNORMAL_BOXMULLER2_ACCURATE,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DLOGNORMAL_BOXMULLER2_ACCURATE</td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SLOGNORMAL_ICDF,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DLOGNORMAL_ICDF,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_SLOGNORMAL_ICDF_ACCURATE,</td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF,</td>
</tr>
<tr>
<td></td>
<td>VSL_METHOD_DLOGNORMAL_ICDF_ACCURATE</td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF,</td>
</tr>
</tbody>
</table>
Method Names for Discrete Distribution Generators

<table>
<thead>
<tr>
<th>RNG</th>
<th>Legacy Method Name</th>
<th>New Method Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>vRngUniform</td>
<td>VSL_METHOD_IUNIFORM_STD</td>
<td>VSL_RNG_METHOD_UNIFORM_STD</td>
</tr>
<tr>
<td>vRngUniformBits</td>
<td>VSL_METHOD_IUNIFORMBITS_STD</td>
<td>VSL_RNG_METHOD_UNIFORMBITS_STD</td>
</tr>
<tr>
<td>vRngBernoulli</td>
<td>VSL_METHOD_IBERNOULLI_ICDF</td>
<td>VSL_RNG_METHOD_BERNOULLI_ICDF</td>
</tr>
<tr>
<td>vRngGeometric</td>
<td>VSL_METHOD_I_GEOMETRIC_ICDF</td>
<td>VSL_RNG_METHOD_GEOMETRIC_ICDF</td>
</tr>
<tr>
<td>vRngBinomial</td>
<td>VSL_METHOD_IBINOMIAL_BTPE</td>
<td>VSL_RNG_METHOD_BINOMIAL_BTPE</td>
</tr>
<tr>
<td>vRngHypergeometric</td>
<td>VSL_METHOD_I_HYPERGEOMETRIC_H2PE</td>
<td>VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE</td>
</tr>
<tr>
<td>vRngPoisson</td>
<td>VSL_METHOD_IPOISSON_PTPE, VSL_METHOD_IPOISSON_POISNORM</td>
<td>VSL_RNG_METHOD_POISSON_PTPE, VSL_RNG_METHOD_POISSON_POISNORM</td>
</tr>
<tr>
<td>vRngPoissonV</td>
<td>VSL_METHOD_IPOISSONV_POISNORM</td>
<td>VSL_RNG_METHOD_POISSONV_POISNORM</td>
</tr>
<tr>
<td>vRngNegBinomial</td>
<td>VSL_METHOD_INEGBINOMIAL_NBAR</td>
<td>VSL_RNG_METHOD_NEGBINOMIAL_NBAR</td>
</tr>
</tbody>
</table>

Continuous Distributions
This section describes routines for generating random numbers with continuous distribution.

vRngUniform Continuous Distribution Generators
Generates random numbers with uniform distribution.

Syntax

```
status = vsRngUniform( method, stream, n, r, a, b );
status = vdRngUniform( method, stream, n, r, a, b );
```

Include Files
- mkl.h

Description
The vRngUniform function generates random numbers uniformly distributed over the interval \([a, b]\), where \(a, b \in \mathbb{R}; a < b\).

The probability density function is given by:
The cumulative distribution function is as follows:

\[ f_{a,b}(x) = \begin{cases} \frac{1}{b - a}, & x \in [a, b), \quad -\infty < x < +\infty \\ 0, & x \notin [a, b) \end{cases} \]

The cumulative distribution function is as follows:

\[ f_{a,b}(x) = \begin{cases} 0, & x < a \\ \frac{x - a}{b - a}, & a \leq x < b, \quad -\infty < x < +\infty \\ 1, & x \geq b \end{cases} \]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method; the specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_UNIFORM_STD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_UNIFORM_STD_ACCURATE</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated.</td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngUniform</td>
<td>Left bound a.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngUniform</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>const float for vsRngUniform</td>
<td>Right bound b.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngUniform</td>
<td></td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngUniform</td>
<td>Vector of ( n ) random numbers uniformly distributed over the interval ([a, b])</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngUniform</td>
<td></td>
</tr>
</tbody>
</table>

Return Values

- VSL_ERROR_OK, VSL_STATUS_OK: Indicates no error, execution is successful.
- VSL_ERROR_NULL_PTR: Stream is a NULL pointer.
- VSL_RNG_ERROR_BAD_STREAM: Stream is not a valid random stream.
- VSL_RNG_ERROR_BAD_UPDATE: Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > n_{\text{max}} \).
- VSL_RNG_ERROR_NO_NUMBERS: Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED: Period of the generator has been exceeded.
- VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED: Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
- VSL_RNG_ERROR_AR5_NOT_SUPPORTED: ARS-5 random number generator is not supported on the CPU running the application.

\textbf{vRngGaussian}

Generates normally distributed random numbers.

Syntax

\[
\text{status} = \text{vsRngGaussian}(\text{method}, \text{stream}, \text{n}, \text{r}, \text{a}, \text{sigma});
\]
\[
\text{status} = \text{vdRngGaussian}(\text{method}, \text{stream}, \text{n}, \text{r}, \text{a}, \text{sigma});
\]

Include Files

- mkl.h

Description

The \text{vRngGaussian} function generates random numbers with normal (Gaussian) distribution with mean value \( a \) and standard deviation \( \sigma \), where

\( a, \sigma \in \mathbb{R} ; \sigma > 0 \).

The probability density function is given by:

\[
f_{a, \sigma}(x) = \frac{1}{\sigma \sqrt{2\pi}} \exp \left( -\frac{(x - a)^2}{2\sigma^2} \right), \quad -\infty < x < +\infty.
\]

The cumulative distribution function is as follows:
The cumulative distribution function \( F_{a,\sigma}(x) \) can be expressed in terms of standard normal distribution \( \Phi(x) \) as

\[
F_{a,\sigma}(x) = \Phi \left( \frac{x - a}{\sigma} \right), \quad -\infty < x < +\infty.
\]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIAN_BOXMULLER2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIAN_ICDF</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated.</td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngGaussian const double for vdRngGaussian</td>
<td>Mean value ( a ).</td>
</tr>
</tbody>
</table>

See brief description of the methods BOXMULLER, BOXMULLER2, and ICDF in Table "Values of <method> in method parameter"
Name | Type | Description
--- | --- | ---
\(\sigma\) | \text{const float} | for \text{vsRngGaussian}
\text{const double} | for \text{vdRngGaussian}

\textbf{Output Parameters}

Name | Type | Description
--- | --- | ---
r | \text{float*} | for \text{vsRngGaussian}
\text{double*} | for \text{vdRngGaussian}

\textbf{Return Values}

- \text{VSL\_ERROR\_OK, VSL\_STATUS\_OK}
  - Indicates no error, execution is successful.
- \text{VSL\_ERROR\_NULL\_PTR}
  - \text{stream} is a NULL pointer.
- \text{VSL\_RNG\_ERROR\_BAD\_STREAM}
  - \text{stream} is not a valid random stream.
- \text{VSL\_RNG\_ERROR\_BAD\_UPDATE}
  - Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> nmax\).
- \text{VSL\_RNG\_ERROR\_NO\_NUMBERS}
  - Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- \text{VSL\_RNG\_ERROR\_QRNG\_PERIOD\_ELAPSED}
  - Period of the generator has been exceeded.
- \text{VSL\_RNG\_ERROR\_NONDETERM\_NRETRIES\_EXCEEDED}
  - Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
- \text{VSL\_RNG\_ERROR\_ARS5\_NOT\_SUPPORTED}
  - ARS-5 random number generator is not supported on the CPU running the application.

\textbf{vRngGaussianMV}

\textit{Generates random numbers from multivariate normal distribution.}

\textbf{Syntax}

\begin{align*}
\text{status} &= \text{vsRngGaussianMV}(\text{method, stream, n, r, dimen, mstorage, a, t }); \\
\text{status} &= \text{vdRngGaussianMV}(\text{method, stream, n, r, dimen, mstorage, a, t });
\end{align*}

\textbf{Include Files}

- mkl.h
**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIANMV_BOXMULLER2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GAUSSIANMV_ICDF</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of (d)-dimensional vectors to be generated</td>
</tr>
<tr>
<td>dimen</td>
<td>const MKL_INT</td>
<td>Dimension (d) ((d \geq 1)) of output random vectors</td>
</tr>
<tr>
<td>mstorage</td>
<td>const MKL_INT</td>
<td>Matrix storage scheme for lower triangular matrix (T). The routine supports three matrix storage schemes:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VSL_MATRIX_STORAGE_FULL— all (d \times d) elements of the matrix (T) are passed, however, only the lower triangle part is actually used in the routine.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VSL_MATRIX_STORAGE_PACKED— lower triangle elements of (T) are packed by rows into a one-dimensional array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VSL_MATRIX_STORAGE_DIAGONAL— only diagonal elements of (T) are passed.</td>
</tr>
<tr>
<td>a</td>
<td>const float*</td>
<td>Mean vector (a) of dimension (d)</td>
</tr>
<tr>
<td></td>
<td>for vsRngGaussianMV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdRngGaussianMV</td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>const float*</td>
<td>Elements of the lower triangular matrix passed according to the matrix (T) storage scheme (mstorage).</td>
</tr>
<tr>
<td></td>
<td>for vsRngGaussianMV</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>for vdRngGaussianMV</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngGaussianMV</td>
<td>Array of (n) random vectors of dimension (dimen)</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngGaussianMV</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The `vRngGaussianMV` function generates random numbers with \(d\)-variate normal (Gaussian) distribution with mean value \(a\) and variance-covariance matrix \(C\), where \(a \in \mathbb{R}^d\); \(C\) is a \(d \times d\) symmetric positive-definite matrix. The probability density function is given by:
where $x \in \mathbb{R}^d$.

Matrix $C$ can be represented as $C = TT^T$, where $T$ is a lower triangular matrix - Cholesky factor of $C$.

Instead of variance-covariance matrix $C$ the generation routines require Cholesky factor of $C$ in input. To compute Cholesky factor of matrix $C$, the user may call Intel® MKL LAPACK routines for matrix factorization: `?potrf` or `?pptrf` for `v?RngGaussianMV/v?rnggaussianmv` routines (? means either `s` or `d` for single and double precision respectively). See Application Notes for more details.

### Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Application Notes

Since matrices are stored in Fortran by columns, while in C they are stored by rows, the usage of Intel® MKL factorization routines (assuming Fortran matrices storage) in combination with multivariate normal RNG (assuming C matrix storage) is slightly different in C and Fortran. The following tables help in using these routines in C and Fortran. For further information please refer to the appropriate VS example file.

#### Using Cholesky Factorization Routines in C

<table>
<thead>
<tr>
<th>Matrix Storage Scheme</th>
<th>Variance-Covariance Matrix Argument</th>
<th>Factorization Routine</th>
<th>UPLO Parameter in Factorization Routine</th>
<th>Result of Factorization as Input Argument for RNG</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_MATRIX_STORAGE_FULL</td>
<td>$C$ in C two-dimensional array</td>
<td><code>spotrf</code> for <code>vsRngGaussianMV</code></td>
<td><code>U</code></td>
<td>Lower triangle of $T$. Upper triangle is not used.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dpotr</code> for <code>vdRngGaussianMV</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VSL_MATRIX_STORAGE_PACKED</td>
<td>Lower triangle of $C$ packed by columns into one-dimensional array</td>
<td><code>spotr</code> for <code>vsRngGaussianMV</code></td>
<td><code>L</code></td>
<td>Lower triangle of $T$ packed by columns into a one-dimensional array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>dpotr</code> for <code>vdRngGaussianMV</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Return Values

`VSL_ERROR_OK, VSL_STATUS_OK` Indicates no error, execution is successful.
VSL_ERROR_NULL_PTR
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM
stream is not a valid random stream.

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, $< 0$ or $> n_{\text{max}}$.

VSL_RNG_ERROR_BAD_UPDATE
Callback function for an abstract BRNG returns $0$ as the number of updated entries in a buffer.

VSL_RNG_ERROR_NO_NUMBERS
Period of the generator has been exceeded.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED
ARS-5 random number generator is not supported on the CPU running the application.

\textbf{vRngExponential}
\textit{Generates exponentially distributed random numbers.}

\textbf{Syntax}

\begin{verbatim}
status = vsRngExponential( method, stream, n, r, a, beta );
status = vdRngExponential( method, stream, n, r, a, beta );
\end{verbatim}

\textbf{Include Files}

- mkl.h

\textbf{Input Parameters}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{method}</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{VSL_RNG_METHOD_EXPONENTIAL_ICDF}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{VSL_RNG_METHOD_EXPONENTIAL_ICDF_ACCURATE}</td>
</tr>
<tr>
<td>\textit{stream}</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>\textit{n}</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>\textit{a}</td>
<td>const float for \texttt{vsRngExponential}</td>
<td>Displacement $a$</td>
</tr>
<tr>
<td>\textit{beta}</td>
<td>const float for \texttt{vsRngExponential}</td>
<td>Scalefactor $\beta$.</td>
</tr>
</tbody>
</table>
const double for vdRngExponential

Output Parameters

Name | Type                  | Description
-----|-----------------------|-------------
    | float* for vsRngExponential | Vector of n exponentially distributed random numbers
    | double* for vdRngExponential |

Description

The `vRngExponential` function generates random numbers with exponential distribution that has displacement $a$ and scalefactor $\beta$, where $a, \beta \in \mathbb{R}; \beta > 0$.

The probability density function is given by:

$$F_{a,\beta}(x) = \begin{cases} \frac{1}{\beta} \exp \left( \frac{-(x-a)}{\beta} \right), & x \geq a, \quad -\infty < x < +\infty, \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp \left( \frac{-(x-a)}{\beta} \right), & x \geq a, \quad -\infty < x < +\infty, \\ 0, & x < a \end{cases}$$

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- `VSL_ERROR_OK, VSL_STATUS_OK`  
  Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`  
  `stream` is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`  
  `stream` is not a valid random stream.
- `VSL_RNG_ERROR_BAD_UPDATE`  
  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
VSL_RNG_ERROR_NO_NUMBERS
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED
ARS-5 random number generator is not supported on the CPU running the application.

vRngLaplace
Generates random numbers with Laplace distribution.

Syntax
status = vsRngLaplace( method, stream, n, r, a, beta);
status = vdRngLaplace( method, stream, n, r, a, beta);

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngLaplace</td>
<td>Mean value $a$</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngLaplace</td>
<td></td>
</tr>
<tr>
<td>beta</td>
<td>const float for vsRngLaplace</td>
<td>Scalefactor $\beta$.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngLaplace</td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngLaplace</td>
<td>Vector of $n$ Laplace distributed random numbers</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngLaplace</td>
<td></td>
</tr>
</tbody>
</table>

Description
The vRngLaplace function generates random numbers with Laplace distribution with mean value (or average) $a$ and scalefactor $\beta$, where $a, \beta \in \mathbb{R}; \beta > 0$. The scalefactor value determines the standard deviation as
The probability density function is given by:

\[ f_{\alpha, \beta}(x) = \frac{1}{\sqrt{2\beta}} \exp\left(-\frac{|x - \alpha|}{\beta}\right), \quad -\infty < x < +\infty. \]

The cumulative distribution function is as follows:

\[ f_{\alpha, \beta}(x) = \begin{cases} 
\frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x \geq \alpha \\
1 - \frac{1}{2} \exp\left(-\frac{|x - \alpha|}{\beta}\right), & x < \alpha 
\end{cases} \quad , \quad -\infty < x < +\infty. \]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- **VSL_ERROR_OK, VSL_STATUS_OK**
  Indicates no error, execution is successful.

  *Stream is a NULL pointer.*

- **VSL_ERROR_NULL_PTR**
  Stream is not a valid random stream.

  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> n_{\text{max}}\).

- **VSL_RNG_ERROR_BAD_STREAM**
  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

  Period of the generator has been exceeded.

  Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

- **VSL_RNG_ERROR_AR5_NOT_SUPPORTED**
  ARS-5 random number generator is not supported on the CPU running the application.
**vRngWeibull**
*Generates Weibull distributed random numbers.*

**Syntax**

```c
status = vsRngWeibull( method, stream, n, r, alpha, a, beta );
status = vdRngWeibull( method, stream, n, r, alpha, a, beta );
```

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>method</code></td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_WEIBULL_ICDF_ACCURATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inverse cumulative distribution function method</td>
</tr>
<tr>
<td><code>stream</code></td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td><code>n</code></td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td><code>alpha</code></td>
<td>const float for vsRngWeibull</td>
<td>Shape α.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngWeibull</td>
<td></td>
</tr>
<tr>
<td><code>a</code></td>
<td>const float for vsRngWeibull</td>
<td>Displacement a.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngWeibull</td>
<td></td>
</tr>
<tr>
<td><code>beta</code></td>
<td>const float for vsRngWeibull</td>
<td>Scalefactor β.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngWeibull</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code></td>
<td>float* for vsRngWeibull</td>
<td>Vector of n Weibull distributed random numbers</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngWeibull</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The `vRngWeibull` function generates Weibull distributed random numbers with displacement $a$, scalefactor $\beta$, and shape $\alpha$, where $\alpha$, $\beta$, $a \in \mathbb{R}$ : $\alpha > 0$, $\beta > 0$.

The probability density function is given by:
The cumulative distribution function is as follows:

\[
F_{\alpha,\beta}(x) = \begin{cases} 
\frac{1}{\beta^\alpha} (x - a)^{\alpha-1} \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a \\
0, & x < a
\end{cases}
\]

The cumulative distribution function is as follows:

\[
F_{\alpha,\beta}(x^*) = \begin{cases} 
1 - \exp\left(-\left(\frac{x-a}{\beta}\right)^\alpha\right), & x \geq a, \ -\infty < x < +\infty \\
0, & x < a
\end{cases}
\]

---

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

---

**Return Values**

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**: 
  stream is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**: 
  stream is not a valid random stream.
- **VSL_RNG_ERROR_BAD_UPDATE**: Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.
- **VSL_RNG_ERROR_NO_NUMBERS**: Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**: Period of the generator has been exceeded.
- **VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED**: Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
- **VSL_RNG_ERROR_ARSG_NOT_SUPPORTED**: ARS-5 random number generator is not supported on the CPU running the application.

vRngCauchy

Generates Cauchy distributed random values.
Syntax

\[
\text{status} = \text{vsRngCauchy}(\text{method}, \text{stream}, n, r, a, beta); \\
\text{status} = \text{vdRngCauchy}(\text{method}, \text{stream}, n, r, a, beta);
\]

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngCauchy</td>
<td>Displacement $a$.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngCauchy</td>
<td></td>
</tr>
<tr>
<td>beta</td>
<td>const float for vsRngCauchy</td>
<td>Scalefactor $\beta$.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngCauchy</td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngCauchy</td>
<td>Vector of $n$ Cauchy distributed random numbers</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngCauchy</td>
<td></td>
</tr>
</tbody>
</table>

Description

The function generates Cauchy distributed random numbers with displacement $a$ and scalefactor $\beta$, where $a, \beta \in \mathbb{R} ; \beta > 0$.

The probability density function is given by:

\[
F_{a,\beta}(x) = \frac{1}{\pi \beta} \left( 1 + \left( \frac{x - a}{\beta} \right)^2 \right), -\infty < x < +\infty.
\]

The cumulative distribution function is as follows:

\[
F_{a,\beta}(x) = \frac{1}{2} + \frac{1}{\pi} \arctan \left( \frac{x - a}{\beta} \right), -\infty < x < +\infty.
\]
Return Values

VSL_ERROR_OK, VSL_STATUS_OK
Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM
stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE
Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \(> n_{\text{max}} \).

VSL_RNG_ERROR_NO_NUMBERS
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Period of the generator has been exceeded.

VSL_RNG_ERROR_NDETERM_NRETRIES_EXCEEDED
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_AR5_NOT_SUPPORTED
ARS-5 random number generator is not supported on the CPU running the application.

vRngRayleigh
Generates Rayleigh distributed random values.

Syntax

\[
\text{status} = \text{vsRngRayleigh}( \text{method}, \text{stream}, n, r, a, \beta ); \\
\text{status} = \text{vdRngRayleigh}( \text{method}, \text{stream}, n, r, a, \beta );
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_RAYLEIGH_ICDF_ACCURATE</td>
</tr>
</tbody>
</table>

Inverse cumulative distribution function method.
**Name** | **Type** | **Description**  
--- | --- | ---  
stream | VSLStreamStatePtr | Pointer to the stream state structure  
n | const MKL_INT | Number of random values to be generated  
a | const float for vsRngRayleigh | Displacement $a$  
 | const double for vdRngRayleigh |  
beta | const float for vsRngRayleigh | Scalefactor $\beta$.  
 | const double for vdRngRayleigh |  

**Output Parameters**  
**Name** | **Type** | **Description**  
--- | --- | ---  
r | float* for vsRngRayleigh | Vector of $n$ Rayleigh distributed random numbers  
 | double* for vdRngRayleigh |  

**Description**  
The `vRngRayleigh` function generates Rayleigh distributed random numbers with displacement $a$ and scalefactor $\beta$, where $a, \beta \in \mathbb{R}; \beta > 0$.  
The Rayleigh distribution is a special case of the Weibull distribution, where the shape parameter $\alpha = 2$.  
The probability density function is given by:

$$f_{a,\beta}(x) = \begin{cases} \frac{2}{\beta^2} \left( \frac{x-a}{\beta^2} \right) \exp \left( - \left( \frac{x-a}{\beta^2} \right) \right), & x \geq a, \ -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

The cumulative distribution function is as follows:

$$F_{a,\beta}(x) = \begin{cases} 1 - \exp \left( - \left( \frac{x-a}{\beta^2} \right) \right), & x \geq a, \ -\infty < x < +\infty. \\ 0, & x < a \end{cases}$$

**Optimization Notice**  
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
Return Values

VSL_ERROR_OK, VSL_STATUS_OK  
Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR  
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM  
stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE  
Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> nmax\).

VSL_RNG_ERROR_NO_NUMBERS  
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED  
Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED  
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED  
ARS-5 random number generator is not supported on the CPU running the application.

vRngLognormal

Generates lognormally distributed random numbers.

Syntax

\[
\text{status} = \text{vsRngLognormal} ( \text{method}, \text{stream}, n, r, a, \text{sigma}, b, \text{beta} );
\]

\[
\text{status} = \text{vdRngLognormal} ( \text{method}, \text{stream}, n, r, a, \text{sigma}, b, \text{beta} );
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_LOGNORMAL_BOXMULLER2_ACCURATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Box Muller 2 based method</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_LOGNORMAL_ICDF_ACCURATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inverse cumulative distribution function based method</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngLognormal</td>
<td>Average ( a ) of the subject normal distribution</td>
</tr>
<tr>
<td></td>
<td>const double for vdrngLognormal</td>
<td></td>
</tr>
<tr>
<td>sigma</td>
<td>const float for vsRngLognormal</td>
<td>Standard deviation ( \sigma ) of the subject normal distribution</td>
</tr>
<tr>
<td></td>
<td>const double for vdrngLognormal</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>const float for vsRngLognormal</td>
<td>Displacement ( b )</td>
</tr>
<tr>
<td></td>
<td>const double for vdrngLognormal</td>
<td></td>
</tr>
<tr>
<td>beta</td>
<td>const float for vsRngLognormal</td>
<td>Scalefactor ( \beta ).</td>
</tr>
<tr>
<td></td>
<td>const double for vdrngLognormal</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngLognormal</td>
<td>Vector of ( n ) lognormally distributed random numbers</td>
</tr>
<tr>
<td></td>
<td>double* for vdrngLognormal</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

The `vRngLognormal` function generates lognormally distributed random numbers with average of distribution \( a \) and standard deviation \( \sigma \) of subject normal distribution, displacement \( b \), and scalefactor \( \beta \), where \( a, \sigma, b, \beta \in \mathbb{R} \); \( \sigma > 0 \), \( \beta > 0 \).

The probability density function is given by:

\[
 f_{a,\sigma,b,\beta}(x) = \begin{cases} 
  \frac{1}{\sigma(x - b)\sqrt{2\pi}} \exp \left( - \frac{\ln((x - b)/\beta) - a}{2\sigma^2} \right), & x > b \\
  0, & x \leq b 
\end{cases}
\]

The cumulative distribution function is as follows:

\[
 F_{a,\sigma,b,\beta}(x) = \begin{cases} 
  \Phi \left( \frac{\ln((x - b)/\beta) - a}{\sigma} \right), & x > b \\
  0, & x \leq b 
\end{cases}
\]
Return Values

VSL_ERROR_OK, VSL_STATUS_OK

Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR

stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM

stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> nmax\).

VSL_RNG_ERROR_NO_NUMBERS

Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED

Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED

Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED

ARS-5 random number generator is not supported on the CPU running the application.

vRngGumbel

Generates Gumbel distributed random values.

Syntax

\[
\begin{align*}
\text{status} &= \text{vsRngGumbel}(\text{method}, \text{stream}, n, r, a, \beta); \\
\text{status} &= \text{vdRngGumbel}(\text{method}, \text{stream}, n, r, a, \beta);
\end{align*}
\]

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GUMBEL_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inverse cumulative distribution function method</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>-------------</td>
</tr>
<tr>
<td>( n )</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>( a )</td>
<td>const float for vsRngGumbel, const double for vdRngGumbel</td>
<td>Displacement ( a ).</td>
</tr>
<tr>
<td>( \beta )</td>
<td>const float for vsRngGumbel, const double for vdRngGumbel</td>
<td>Scalefactor ( \beta ).</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r )</td>
<td>float* for vsRngGumbel, double* for vdRngGumbel</td>
<td>Vector of ( n ) random numbers with Gumbel distribution</td>
</tr>
</tbody>
</table>

**Description**

The \( \text{vRngGumbel} \) function generates Gumbel distributed random numbers with displacement \( a \) and scalefactor \( \beta \), where \( a, \beta \in \mathbb{R}; \beta > 0 \).

The probability density function is given by:

\[
 f_{a,\beta}(x) = \frac{1}{\beta} \exp \left( \frac{x - a}{\beta} \right) \exp \left( - \exp \left( \frac{x - a}{\beta} \right) \right), \quad -\infty < x < +\infty.
\]

The cumulative distribution function is as follows:

\[
 F_{a,\beta}(x) = 1 - \exp \left( - \exp \left( \frac{x - a}{\beta} \right) \right), \quad -\infty < x < +\infty
\]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- \( \text{VSL\_ERROR\_OK}, \text{VSL\_STATUS\_OK} \): Indicates no error, execution is successful.
- \( \text{VSL\_ERROR\_NULL\_PTR} \): \( stream \) is a NULL pointer.
- \( \text{VSL\_RNG\_ERROR\_BAD\_STREAM} \): \( stream \) is not a valid random stream.
VSL_RNG_ERROR_BAD_UPDATE  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, < 0 or > nmax.

VSL_RNG_ERROR_NO_NUMBERS  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED  Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED  Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARSG_NOT_SUPPORTED  ARS-5 random number generator is not supported on the CPU running the application.

vRngGamma
Generates gamma distributed random values.

Syntax

\[
\text{status} = \text{vsRngGamma( method, stream, n, r, alpha, a, beta );}
\]
\[
\text{status} = \text{vdRngGamma( method, stream, n, r, alpha, a, beta );}
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{method}</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td>\text{stream}</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>\text{n}</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>\text{alpha}</td>
<td>const float for vsRngGamma</td>
<td>Shape ( \alpha ).</td>
</tr>
<tr>
<td>\text{a}</td>
<td>const float for vsRngGamma</td>
<td>Displacement ( a ).</td>
</tr>
<tr>
<td>\text{beta}</td>
<td>const float for vsRngGamma</td>
<td>Scalefactor ( \beta ).</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>float* for vsRngGamma</td>
<td>Vector of $n$ random numbers with gamma distribution</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngGamma</td>
<td></td>
</tr>
</tbody>
</table>

Description

The `vRngGamma` function generates random numbers with gamma distribution that has shape parameter $\alpha$, displacement $a$, and scale parameter $\beta$, where $\alpha$, $\beta$, and $a \in \mathbb{R}$; $\alpha > 0$, $\beta > 0$.

The probability density function is given by:

$$f_{\alpha,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} (x-a)^{\alpha-1}e^{-(x-a)/\beta}, & x \geq a, -\infty < x < +\infty \\ 0, & x < a \end{cases}$$

where $\Gamma(\alpha)$ is the complete gamma function.

The cumulative distribution function is as follows:

$$F_{\alpha,\beta}(x) = \begin{cases} \frac{1}{\Gamma(\alpha)\beta^\alpha} \int_a^x (y-a)^{\alpha-1}e^{-(y-a)/\beta}dy, & x \geq a, -\infty < x < +\infty \\ 0, & x < a \end{cases}$$

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- `VSL_ERROR_OK`, `VSL_STATUS_OK`  
  Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`  
  Stream is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`  
  Stream is not a valid random stream.
- `VSL_RNG_ERROR_BAD_UPDATE`  
  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, $< 0$ or $> nmax$.
- `VSL_RNG_ERROR_NO_NUMBERS`  
  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`  
  Period of the generator has been exceeded.
VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED

Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED

ARS-5 random number generator is not supported on the CPU running the application.

vRngBeta

Generates beta distributed random values.

Syntax

```c
status = vsRngBeta( method, stream, n, r, p, q, a, beta );
status = vdRngBeta( method, stream, n, r, p, q, a, beta );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_BETA_CJA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_BETA_CJA_ACCURATE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See brief description of the method CJA in Table &quot;Values of &lt;method&gt; in method parameter”</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>p</td>
<td>const float for vsRngBeta</td>
<td>Shape p</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngBeta</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>const float for vsRngBeta</td>
<td>Shape q</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngBeta</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>const float for vsRngBeta</td>
<td>Displacementa.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngBeta</td>
<td></td>
</tr>
<tr>
<td>beta</td>
<td>const float for vsRngBeta</td>
<td>Scalefactor β.</td>
</tr>
<tr>
<td></td>
<td>const double for vdRngBeta</td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngBeta</td>
<td>Vector of n random numbers with beta distribution</td>
</tr>
<tr>
<td></td>
<td>double* for vdRngBeta</td>
<td></td>
</tr>
</tbody>
</table>
**Description**

The vRngBeta function generates random numbers with beta distribution that has shape parameters \( p \) and \( q \), displacement \( a \), and scale parameter \( \beta \), where \( p, q, a, \) and \( \beta \in \mathbb{R} ; p > 0, q > 0, \beta > 0 \).

The probability density function is given by:

\[
 f_{p,q,a,\beta}(x) = \begin{cases} 
 \frac{1}{B(p, q) \beta^{p+q-1}} (x - a)^{p-1} (\beta + a - x)^{q-1}, & a \leq x < a + \beta, \quad -\infty < x < \infty, \\
 0, & x < a, \quad x \geq a + \beta 
\end{cases}
\]

where \( B(p, q) \) is the complete beta function.

The cumulative distribution function is as follows:

\[
 F_{p,q,a,\beta}(x) = \begin{cases} 
 0, & x < a \\
 \frac{1}{B(p, q) \beta^{p+q-1}} \int_a^x [y - a]^{p-1} [(\beta + a - y)^{q-1} - \beta y^{q-1}], & a \leq x < a + \beta, \quad -\infty < x < \infty, \\
 1, & x \geq a + \beta 
\end{cases}
\]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- **VSL_ERROR_OK, VSL_STATUS_OK**
  Indicates no error, execution is successful.

- **VSL_ERROR_NULL_PTR**
  \emph{stream} is a NULL pointer.

- **VSL_RNG_ERROR_BAD_STREAM**
  \emph{stream} is not a valid random stream.

- **VSL_RNG_ERROR_BAD_UPDATE**
  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > n_{\text{max}} \).

- **VSL_RNG_ERROR_NO_NUMBERS**
  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**
  Period of the generator has been exceeded.

- **VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED**
  Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

- **VSL_RNG_ERROR_ARSG5_NOT_SUPPORTED**
  ARS-5 random number generator is not supported on the CPU running the application.
vRngChiSquare
Generates chi-square distributed random values.

Syntax
status = vsRngChiSquare( method, stream, n, r, v );
status = vdRngChiSquare( method, stream, n, r, v );

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is: VSL_RNG_METHOD_CHISQUARE_CHI2GAMMA</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>v</td>
<td>const MKL_INT</td>
<td>Degrees of freedom</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>float* for vsRngChiSquare double* for vdRngChiSquare</td>
<td>Vector of n random numbers with chi-square distribution</td>
</tr>
</tbody>
</table>

Description
The vRngChiSquare function generates random numbers with chi-square distribution and $v$ degrees of freedom, $v \in \mathbb{N}$, $v > 0$.

The probability density function is:

$$f_v(x) = \begin{cases} \frac{x^{v-2/2} e^{-x/2}}{2^{v/2} \Gamma(v/2)}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$

The cumulative distribution function is:

$$F_v(x) = \begin{cases} \frac{x^{v-2/2} - y^{v-2/2}}{2^{v/2} \Gamma(v/2)}, & x \geq 0 \\ 0, & x < 0 \end{cases}$$
Return Values

VSL_ERROR_OK, VSL_STATUS_OK
Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM
stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE
Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> n_{\text{max}}\).

VSL_RNG_ERROR_NO_NUMBERS
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED
ARS-5 random number generator is not supported on the CPU running the application.

Discrete Distributions
This section describes routines for generating random numbers with discrete distribution.

vRngUniform Discrete Distribution Generators
Generates random numbers uniformly distributed over the interval \([a, b)\).

Syntax

```c
status = viRngUniform( method, stream, n, r, a, b );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method; the specific value is as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_UNIFORM_STD</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Standard method. Currently there is only one method for this distribution generator.</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>a</td>
<td>const int</td>
<td>Left interval bound (a)</td>
</tr>
<tr>
<td>b</td>
<td>const int</td>
<td>Right interval bound (b)</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of ( n ) random numbers uniformly distributed over the interval ([a, b])</td>
</tr>
</tbody>
</table>

Description

The vRngUniform function generates random numbers uniformly distributed over the interval \([a, b]\), where \( a, b \) are the left and right bounds of the interval respectively, and \( a, b \in \mathbb{Z}; a < b \).

The probability distribution is given by:

\[
P(X = k) = \frac{1}{b - a}, \; k \in \{a, a + 1, \ldots, b - 1\}.
\]

The cumulative distribution function is as follows:

\[
F_{a,b}(x) = \begin{cases} 
0, & x < a \\
\frac{x - a + 1}{b - a}, & a \leq x < b, x \in \mathbb{R} \\
1, & x \geq b
\end{cases}
\]

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**: Indicates no error, execution is successful.
- **VSL_ERROR_NULL_PTR**: \( \text{stream} \) is a NULL pointer.
- **VSL_RNG_ERROR_BAD_STREAM**: \( \text{stream} \) is not a valid random stream.
- **VSL_RNG_ERROR_BAD_UPDATE**: Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > n_{max} \).
- **VSL_RNG_ERROR_NO_NUMBERS**: Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**: Period of the generator has been exceeded.
**VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED**

Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

**VSL_RNG_ERROR_ARS5_NOT_SUPPORTED**

ARS-5 random number generator is not supported on the CPU running the application.

**vRngUniformBits**

*Generates bits of underlying BRNG integer recurrence.*

**Syntax**

```c
status = viRngUniformBits( method, stream, n, r );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method; the specific value is <code>VSL_RNG_METHOD_UNIFORMBITS_STD</code></td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>unsigned int*</td>
<td>Vector of n random integer numbers. If the <code>stream</code> was generated by a 64 or a 128-bit generator, each integer value is represented by two or four elements of <code>r</code> respectively. The number of bytes occupied by each integer is contained in the field <code>WordSize</code> of the structure <code>VSLBRngProperties</code>. The total number of bits that are actually used to store the value are contained in the field <code>NBits</code> of the same structure. See <a href="#">Advanced Service Routines</a> for a more detailed discussion of <code>VSLBRngProperties</code>.</td>
</tr>
</tbody>
</table>

**Description**

The `vRngUniformBits` function generates integer random values with uniform bit distribution. The generators of uniformly distributed numbers can be represented as recurrence relations over integer values in modular arithmetic. Apparently, each integer can be treated as a vector of several bits. In a truly random generator, these bits are random, while in pseudorandom generators this randomness can be violated. For example, a well known drawback of linear congruential generators is that lower bits are less random than higher bits (for example, see [Knuth81]). For this reason, care should be taken when using this function. Typically, in a 32-bit LCG only 24 higher bits of an integer value can be considered random. See [VS Notes](#) for details.
Return Values

VSL_ERROR_OK, VSL_STATUS_OK  
Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR  
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM  
stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE  
Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \text{ or } > nmax\).

VSL_RNG_ERROR_NO_NUMBERS  
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED  
Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRÉTRIES_EXCEED  
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_AR55_NOT_SUPPORTED  
ARS-5 random number generator is not supported on the CPU running the application.

vRngUniformBits32
Generates uniformly distributed bits in 32-bit chunks.

Syntax

status = vIRngUniformBits32( method, stream, n, r );

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method; the specific value is</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>unsigned int*</td>
<td>Vector of ( n ) 32-bit random integer numbers with uniform bit distribution.</td>
</tr>
</tbody>
</table>

Description

The `vRngUniformBits32` function generates uniformly distributed bits in 32-bit chunks. Unlike `vRngUniformBits`, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, `vRngUniformBits32` is designed to ensure each bit in the 32-bit chunk is uniformly distributed. See VS Notes for details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- `VSL_ERROR_OK, VSL_STATUS_OK`: Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`: `stream` is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`: `stream` is not a valid random stream.
- `VSL_RNG_ERROR_BRNG_NOT_SUPPORTED`: BRNG is not supported by the function.
- `VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED`: Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
- `VSL_RNG_ERROR_AR5_NOT_SUPPORTED`: ARS-5 random number generator is not supported on the CPU running the application.

`vRngUniformBits64`

Generates uniformly distributed bits in 64-bit chunks.

Syntax

```c
status = vRngUniformBits64(method, stream, n, r);
```

Include Files

- `mkl.h`
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method; the specific value is</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>unsigned MKL_INT64*</td>
<td>Vector of ( n ) 64-bit random integer numbers with uniform bit distribution.</td>
</tr>
</tbody>
</table>

Description

The \texttt{vRngUniformBits64} function generates uniformly distributed bits in 64-bit chunks. Unlike \texttt{vRngUniformBits}, which provides the output of underlying integer recurrence and does not guarantee uniform distribution across bits, \texttt{vRngUniformBits64} is designed to ensure each bit in the 64-bit chunk is uniformly distributed. See VS Notes for details.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- \texttt{VSL_ERROR_OK}, \texttt{VSL_STATUS_OK} Indicates no error, execution is successful.
- \texttt{VSL_ERROR_NULL_PTR} 
  \textit{stream is a NULL pointer.}
- \texttt{VSL_RNG_ERROR_BAD_STREAM} 
  \textit{stream is not a valid random stream.}
- \texttt{VSL_RNG_ERROR_BRNG_NOT_SUPPORTED} 
  \textit{BRNG is not supported by the function.}
- \texttt{VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED} 
  Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
- \texttt{VSL_RNG_ERROR_AR5_NOT_SUPPORTED} 
  ARS-5 random number generator is not supported on the CPU running the application.
vRngBernoulli

Generates Bernoulli distributed random values.

Syntax

```c
status = viRngBernoulli(method, stream, n, r, p);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_BERNOULLI_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inverse cumulative distribution function method.</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>p</td>
<td>const double</td>
<td>Success probability p of a trial</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of n Bernoulli distributed random values</td>
</tr>
</tbody>
</table>

Description

The `vRngBernoulli` function generates Bernoulli distributed random numbers with probability `p` of a single trial success, where

\[ p \in \mathbb{R}; 0 \leq p \leq 1. \]

A variate is called Bernoulli distributed, if after a trial it is equal to 1 with probability of success `p`, and to 0 with probability `1 - p`.

The probability distribution is given by:

\[ P(X = 1) = p \]
\[ P(X = 0) = 1 - p \]

The cumulative distribution function is as follows:

\[
F_p(x) = \begin{cases} 
0, & x < 0 \\
1 - p, & 0 \leq x < 1, x \in \mathbb{R} \\
1, & x \geq 1
\end{cases}
\]
Return Values

VSL_ERROR_OK, VSL_STATUS_OK
Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR
stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM
stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE
Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > n_{\text{max}} \).

VSL_RNG_ERROR_NO_NUMBERS
Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Period of the generator has been exceeded.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED
ARS-5 random number generator is not supported on the CPU running the application.

vRngGeometric
Generates geometrically distributed random values.

Syntax

\[ \text{status} = \text{viRngGeometric}( \text{method}, \text{stream}, n, r, p ); \]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_GEOMETRIC_ICDF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inverse cumulative distribution function method.</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>p</td>
<td>const double</td>
<td>Success probability ( p ) of a trial</td>
</tr>
</tbody>
</table>

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of ( n ) geometrically distributed random values</td>
</tr>
</tbody>
</table>

### Description

The `vRngGeometric` function generates geometrically distributed random numbers with probability \( p \) of a single trial success, where \( p \in \mathbb{R}; 0 < p < 1 \).

A geometrically distributed variate represents the number of independent Bernoulli trials preceding the first success. The probability of a single Bernoulli trial success is \( p \).

The probability distribution is given by:

\[
P(X = k) = p \cdot (1 - p)^k, \quad k \in \{0,1,2, \ldots \}.
\]

The cumulative distribution function is as follows:

\[
F_p(x) = \begin{cases} 
0, & x < 0 \\
1 - (1 - p)^{\lfloor x+1 \rfloor}, & 0 \geq x \in \mathbb{R}.
\end{cases}
\]

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Return Values

- `VSL_ERROR_OK, VSL_STATUS_OK`: Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`: `stream` is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`: `stream` is not a valid random stream.
- `VSL_RNG_ERROR_BAD_UPDATE`: Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > n_{max} \).
- `VSL_RNG_ERROR_NO_NUMBERS`: Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`: Period of the generator has been exceeded.
- `VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED`: Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
ARS-5 random number generator is not supported on the CPU running the application.

vRngBinomial
Generates binomially distributed random numbers.

Syntax

\[
\text{status} = \text{viRngBinomial}(\text{method, stream, n, r, ntrial, p});
\]

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is as follows:</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>ntrial</td>
<td>const int</td>
<td>Number of independent trials m</td>
</tr>
<tr>
<td>p</td>
<td>const double</td>
<td>Success probability (p) of a single trial</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of (n) binomially distributed random values</td>
</tr>
</tbody>
</table>

Description

The vRngBinomial function generates binomially distributed random numbers with number of independent Bernoulli trials \(m\), and with probability \(p\) of a single trial success, where \(p \in \mathbb{R}; 0 \leq p \leq 1, m \in \mathbb{N}\).

A binomially distributed variate represents the number of successes in \(m\) independent Bernoulli trials with probability of a single trial success \(p\).

The probability distribution is given by:

\[
P(X = k) = \binom{m}{k} p^k (1 - p)^{m-k}, \quad k \in \{0, 1, \ldots, m\}.
\]

The cumulative distribution function is as follows:
\[ F_{m,p}(x) = \begin{cases} 
0, & x < 0 \\
\sum_{k=0}^{1} \binom{k}{m}p^k(1-p)^{m-k}, & 0 \leq x < m, x \in R \\
1, & x \geq m 
\end{cases} \]

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- **VSL_ERROR_OK, VSL_STATUS_OK**
  Indicates no error, execution is successful.
  stream is a NULL pointer.

- **VSL_ERROR_NULL_PTR**
  stream is not a valid random stream.

- **VSL_RNG_ERROR_BAD_STREAM**
  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \( < 0 \) or \( > n_{max} \).

- **VSL_RNG_ERROR_BAD_UPDATE**
  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

- **VSL_RNG_ERROR_NO_NUMBERS**
  Period of the generator has been exceeded.

- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**
  Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

- **VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED**
  ARS-5 random number generator is not supported on the CPU running the application.

**vRngHypergeometric**

*Generates hypergeometrically distributed random values.*

**Syntax**

\[ \text{status} = \text{viRngHypergeometric}( \text{method, stream, n, r, l, s, m} ); \]

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is as follows: VSL_RNG_METHOD_HYPERGEOMETRIC_H2PE. See brief description of the H2PE method in Table &quot;Values of &lt;method&gt; in method parameter&quot;</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>l</td>
<td>const int</td>
<td>Lot size l</td>
</tr>
<tr>
<td>s</td>
<td>const int</td>
<td>Size of sampling without replacement s</td>
</tr>
<tr>
<td>m</td>
<td>const int</td>
<td>Number of marked elements m</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of n hypergeometrically distributed random values</td>
</tr>
</tbody>
</table>

Description

The vRngHypergeometric function generates hypergeometrically distributed random values with lot size l, size of sampling s, and number of marked elements in the lot m, where \( l, m, s \in \mathbb{N} \cup \{0\} \); \( l \geq \max(s, m) \).

Consider a lot of \( l \) elements comprising \( m \) "marked" and \( l-m \) "unmarked" elements. A trial sampling without replacement of exactly \( s \) elements from this lot helps to define the hypergeometric distribution, which is the probability that the group of \( s \) elements contains exactly \( k \) marked elements.

The probability distribution is given by:

\[
P(X = k) = \frac{{C_k^s C_{l-m}^{s-k}}}{{C_l^s}}
\]

, \( k \in \{\max(0, s + m - l), ..., \min(s, m)\} \)

The cumulative distribution function is as follows:

\[
F_{2,\alpha}^{2,\alpha}(x) = \begin{cases} 
0, & x < \max(0, s + m - l) \\
\sum_{x=1}^{\max(0, s + m - l)} \frac{\binom{x}{s} \binom{s-x}{l-m}}{\binom{s}{l}}, & \max(0, s + m - l) \leq x \leq \min(s, m) \\
1, & x > \min(s, m) 
\end{cases}
\]

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel.
**Return Values**

VSL_ERROR_OK, VSL_STATUS_OK
- Indicates no error, execution is successful.

VSL_ERROR_NULL_PTR
- stream is a NULL pointer.

VSL_RNG_ERROR_BAD_STREAM
- stream is not a valid random stream.

VSL_RNG_ERROR_BAD_UPDATE
- Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \(> nmax\).

VSL_RNG_ERROR_NO_NUMBERS
- Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
- Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED
- Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_ARS5_NOT_SUPPORTED
- ARS-5 random number generator is not supported on the CPU running the application.

**vRngPoisson**

*Generates Poisson distributed random values.*

**Syntax**

```c
status = viRngPoisson( method, stream, n, r, lambda );
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific values are as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_POISSON_PTEPE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_POISSON_POISNORM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See brief description of the PTEPE and POISNORM methods in</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table &quot;Values of &lt;method&gt; in method parameter&quot;.</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
lambda | const double | Distribution parameter $\lambda$.

**output parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
r | int* | Vector of $n$ Poisson distributed random values |

**Description**
The \texttt{vRng"Poisson} function generates Poisson distributed random numbers with distribution parameter $\lambda$, where $\lambda \in R; \lambda > 0$.

The probability distribution is given by:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

$k \in \{0, 1, 2, \ldots\}$.

The cumulative distribution function is as follows:

$$F_\lambda(x) = \begin{cases} 
\sum_{k=0}^{\left\lfloor x \right\rfloor} \frac{\lambda^k e^{-\lambda}}{k!}, & x \geq 0, x \in R \\
0, & x < 0 \end{cases}$$

**Optimization Notice**
Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Return Values**

- \texttt{VSL\_ERROR\_OK, VSL\_STATUS\_OK} Indicates no error, execution is successful.
  - \texttt{stream} is a NULL pointer.
- \texttt{VSL\_ERROR\_NULL\_PTR} \texttt{stream} is not a valid random stream.
- \texttt{VSL\_RNG\_ERROR\_BAD\_STREAM} Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, $< 0$ or $> n\text{max}$.
- \texttt{VSL\_RNG\_ERROR\_BAD\_UPDATE} Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED
Period of the generator has been exceeded.

VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED
Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.

VSL_RNG_ERROR_AR5_NOT_SUPPORTED
ARS-5 random number generator is not supported on the CPU running the application.

vRngPoissonV
Generates Poisson distributed random values with varying mean.

Syntax
status = viRngPoissonV( method, stream, n, r, lambda );

Include Files
• mkl.h

Description
The mkl::rng::poisson_v class object is used in the mkl::rng::generate function to provide \( n \) Poisson distributed random numbers \( x_1, \ldots, x_n \) with distribution parameter \( \lambda_i \), where \( \lambda_i \in \mathbb{R}; \lambda_i > 0 \).

The probability distribution is given by:
\[
P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, \quad k \in \{0, 1, 2, \ldots\}.
\]

The cumulative distribution function is as follows:
\[
F_{\lambda_i}(x) = \begin{cases} 
\sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, & x \geq 0, \quad x \in \mathbb{R} \\
0, & x < 0
\end{cases}
\]

Input Parameters
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_POISSONV_POISNORM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See brief description of the POISNORM method in Table &quot;Values of &lt;method&gt; in method parameter&quot;</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>Pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>lambda</td>
<td>const double*</td>
<td>Array of ( n ) distribution parameters ( \lambda_i ).</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of (n) Poisson distributed random values</td>
</tr>
</tbody>
</table>

Description

The `vRngPoissonV` function generates \(n\) Poisson distributed random numbers \(x_i (i = 1, \ldots, n)\) with distribution parameter \(\lambda_i\), where \(\lambda_i \in \mathbb{R}; \lambda_i > 0\).

The probability distribution is given by:

\[
P(X_i = k) = \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, \quad k \in \{0, 1, 2, \ldots\}.
\]

The cumulative distribution function is as follows:

\[
F_{\lambda_i}(x) = \begin{cases} 
\sum_{k=0}^{\lfloor x \rfloor} \frac{\lambda_i^k \exp(-\lambda_i)}{k!}, & x \geq 0, x \in \mathbb{R} \\
0, & x < 0
\end{cases}
\]

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

- `VSL_ERROR_OK, VSL_STATUS_OK`: Indicates no error, execution is successful.
- `VSL_ERROR_NULL_PTR`: \textit{stream} is a NULL pointer.
- `VSL_RNG_ERROR_BAD_STREAM`: \textit{stream} is not a valid random stream.
- `VSL_RNG_ERROR_BAD_UPDATE`: Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0\) or \(> n_{\text{max}}\).
- `VSL_RNG_ERROR_NO_NUMBERS`: Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- `VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED`: Period of the generator has been exceeded.
- `VSL_RNG_ERROR_NONDETERM_NRETRIES_EXCEEDED`: Number of retries to generate a random number by using non-deterministic random number generator exceeds threshold.
ARS-5 random number generator is not supported on the CPU running the application.

**vRngNegBinomial**

Generates random numbers with negative binomial distribution.

**Syntax**

```c
status = viRngNegbinomial( method, stream, n, r, a, p );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Generation method. The specific value is:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>VSL_RNG_METHOD_NEGBINOMIAL_NBAR</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See brief description of the NBAR method in Table &quot;Values of &lt;method&gt; in method parameter&quot;</td>
</tr>
<tr>
<td>stream</td>
<td>VSLStreamStatePtr</td>
<td>pointer to the stream state structure</td>
</tr>
<tr>
<td>n</td>
<td>const MKL_INT</td>
<td>Number of random values to be generated</td>
</tr>
<tr>
<td>a</td>
<td>const double</td>
<td>The first distribution parameter a</td>
</tr>
<tr>
<td>p</td>
<td>const double</td>
<td>The second distribution parameter p</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>int*</td>
<td>Vector of n random values with negative binomial distribution.</td>
</tr>
</tbody>
</table>

**Description**

The vRngNegBinomial function generates random numbers with negative binomial distribution and distribution parameters $a$ and $p$, where $p, a \in \mathbb{R}; 0 < p < 1; a > 0$.

If the first distribution parameter $a \in \mathbb{N}$, this distribution is the same as Pascal distribution. If $a \in \mathbb{N}$, the distribution can be interpreted as the expected time of $a$-th success in a sequence of Bernoulli trials, when the probability of success is $p$.

The probability distribution is given by:

$$P(X = k) = \binom{x+k-1}{k}p^a(1-p)^k, k \in \{0, 1, 2, \ldots\}.$$

The cumulative distribution function is as follows:
\[
F_{a,b}(x) = \begin{cases} 
\binom{k}{x} \sum_{k=0}^{k=x} \binom{n-k}{r-k} b^k (1-p)^{n-k}, & x \geq 0, \\
0, & x < 0
\end{cases}
\]

Return Values

- **VSL_ERROR_OK, VSL_STATUS_OK**
  
  Indicates no error, execution is successful.

- **VSL_ERROR_NULL_PTR**
  
  *stream* is a NULL pointer.

- **VSL_RNG_ERROR_BAD_STREAM**
  
  *stream* is not a valid random stream.

- **VSL_RNG_ERROR_BAD_UPDATE**
  
  Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, \(< 0 \) or \( > nmax \).

- **VSL_RNG_ERROR_NO_NUMBERS**
  
  Callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.

- **VSL_RNG_ERROR_QRNG_PERIOD_ELAPSED**
  
  Period of the generator has been exceeded.

- **VSL_RNG_ERROR_ARSG5_NOT_SUPPORTED**
  
  ARS-5 random number generator is not supported on the CPU running the application.

vRngMultinomial

Generates multinomially distributed random numbers.

Syntax

\[
\text{status} = \text{vRngMultinomial}( \text{method, stream, n, r, ntrial, k, p} );
\]

Include Files

- *mkl.h*

Input Parameters

- **method**
  
  const MKL_INT
  
  Generation method. The specific value is as follows:

  \[
  \text{VSL_RNG_METHOD_MULTINOMIAL_MULTPOISSON}
  \]

- **stream**
  
  VSLStreamStatePtr
Pointer to the stream state structure.

\[ \text{const MKL_INT} \]

Number of random values to be generated.

\( n \)

\[ \text{const int} \]

Number of independent trials \( m \).

\( n_{\text{trial}} \)

\[ \text{const int} \]

Number of possible outcomes.

\( k \)

\[ \text{const double*} \]

Probability vector of \( k \) possible outcomes.

**Output Parameters**

\[ r \]

\[ \text{int*} \]

Array of \( n \) random vectors of dimension \( k \).

**Description**

The `vRngMultinomial` function generates multinomially distributed random numbers with \( m \) independent trials and \( k \) possible mutually exclusive outcomes, with corresponding probabilities \( p_i \), where \( p_i \in \mathbb{R}; 0 \leq p_i \leq 1, m \in \mathbb{N}, k \in \mathbb{N} \).

The probability distribution is given by:

\[
P(X_1 = x_1, \ldots, X_k = x_k) = \frac{m!}{\prod_{i=1}^{k} x_i!} \prod_{i=1}^{k} p_i^{x_i}, \quad 0 \leq x_i \leq m, \sum_{i=1}^{k} x_i = m
\]

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**NOTE**

For information on random number generators used for statistical computations in MKL Vector Statistics, see [MKL Vector Statistics](#).
**Return Values**

- **VSL_ERROR_OK**, **VSL_STATUS_OK**: Indicates no error (execution was successful).
- **VSL_ERROR_NULL_PTR**: stream is a NULL pointer.
- **VSL_RNG_ERROR_BAD_Stream** EAM: stream is not a valid random stream.
- **VSL_RNG_ERROR_BAD_UPDATE**: A callback function for an abstract BRNG returns an invalid number of updated entries in a buffer; that is, \(< 0 \) or \( > \text{nmax} \).
- **VSL_RNG_ERROR_NO_NUMBERS**: A callback function for an abstract BRNG returns 0 as the number of updated entries in a buffer.
- **VSL_RNG_ERROR_ARS5_NOT_SUPPORTED**: An ARS-5 random number generator is not supported on the CPU running the application.
- **VSL_DISTR_MULTINOMIAL_BAD_PROBABILITY_ARRAY**: Bad multinomial distribution probability array.

**Advanced Service Routines**

This section describes service routines for registering a user-designed basic generator (**vslRegisterBrng**) and for obtaining properties of the previously registered basic generators (**vslGetBrngProperties**). See VS Notes ("Basic Generators" section of VS Structure chapter) for substantiation of the need for several basic generators including user-defined BRNGs.

**Advanced Service Routine Data Types**

The Advanced Service routines refer to a structure defining the properties of the basic generator.

This structure is described as follows:

```c
typedef struct _VSLBRngProperties {
    int StreamStateSize;
    int NSeeds;
    int IncludesZero;
    int WordSize;
    int NBits;
    InitStreamPtr InitStream;
    sBRngPtr sBRng;
    dBRngPtr dBRng;
    iBRngPtr iBRng;
} VSLBRngProperties;
```

The following table provides brief descriptions of the fields engaged in the above structure:

<table>
<thead>
<tr>
<th>Field</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StreamStateSize</td>
<td>The size, in bytes, of the stream state structure for a given basic generator.</td>
</tr>
<tr>
<td>NSeeds</td>
<td>The number of 32-bit initial conditions (seeds) necessary to initialize the stream state structure for a given basic generator.</td>
</tr>
<tr>
<td>IncludesZero</td>
<td>Flag value indicating whether the generator can produce a random 0.</td>
</tr>
<tr>
<td>Field</td>
<td>Short Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>WordSize</td>
<td>Machine word size, in bytes, used in integer-value computations. Possible values: 4, 8, and 16 for 32, 64, and 128-bit generators, respectively.</td>
</tr>
<tr>
<td>NBits</td>
<td>The number of bits required to represent a random value in integer arithmetic. Note that, for instance, 48-bit random values are stored to 64-bit (8 byte) memory locations. In this case, ( \text{wordsize}/\text{WordSize} ) is equal to 8 (number of bytes used to store the random value), while ( \text{nbits}/\text{NBits} ) contains the actual number of bits occupied by the value (in this example, 48).</td>
</tr>
<tr>
<td>InitStream</td>
<td>Contains the pointer to the initialization routine of a given basic generator.</td>
</tr>
<tr>
<td>sBRng</td>
<td>Contains the pointer to the basic generator of single precision real numbers uniformly distributed over the interval ((a,b)) (float).</td>
</tr>
<tr>
<td>dBRng</td>
<td>Contains the pointer to the basic generator of double precision real numbers uniformly distributed over the interval ((a,b)) (double).</td>
</tr>
<tr>
<td>iBRng</td>
<td>Contains the pointer to the basic generator of integer numbers with uniform bit distribution (unsigned int).</td>
</tr>
</tbody>
</table>

**vslRegisterBrng**

Registers user-defined basic generator.

**Syntax**

\[
\text{brng} = \text{vslRegisterBrng}( \&\text{properties} );
\]

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties</td>
<td>const VSLBRngProperties*</td>
<td>Pointer to the structure containing properties of the basic generator to be registered</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brng</td>
<td>int</td>
<td>Number (index) of the registered basic generator; used for identification. Negative values indicate the registration error.</td>
</tr>
</tbody>
</table>

**Description**

An example of a registration procedure can be found in the respective directory of the VS examples.

---

1 A specific generator that permits operations over single bits and bit groups of random numbers.
Return Values

VSL_ERROR_OK, VSL_STATUS_OK  
Indicates no error, execution is successful.

VSL_RNG_ERROR_BRNG_TABLE_FULL  
Registration cannot be completed due to lack of free entries in the table of registered BRNGs.

VSL_RNG_ERROR_BAD_STREAM_STATE_SIZE  
Bad value in StreamStateSize field.

VSL_RNG_ERROR_BAD_WORD_SIZE  
Bad value in WordSize field.

VSL_RNG_ERROR_BAD_NSEEDS  
Bad value in NSeeds field.

VSL_RNG_ERROR_BAD_NBITS  
Bad value in NBits field.

VSL_ERROR_NULL_PTR  
At least one of the fields iBrng, dBrng, sBrng or InitStream is a NULL pointer.

vslGetBrngProperties

*Returns structure with properties of a given basic generator.*

Syntax

```c
status = vslGetBrngProperties( brng, &properties );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>brng</td>
<td>const int</td>
<td>Number (index) of the registered basic generator; used for identification. See specific values in Table &quot;Values of brng parameter&quot;. Negative values indicate the registration error.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>properties</td>
<td>VSLBRngProperties*</td>
<td>Pointer to the structure containing properties of the generator with number brng</td>
</tr>
</tbody>
</table>

Description

The vslGetBrngProperties function returns a structure with properties of a given basic generator.

Return Values

VSL_ERROR_OK, VSL_STATUS_OK  
Indicates no error, execution is successful.

VSL_RNG_ERROR_INVALID_BRNG_INDEX  
BRNG index is invalid.
Formats for User-Designed Generators

To register a user-designed basic generator using `vslRegisterBrng` function, you need to pass the pointer `iBrng` to the integer-value implementation of the generator; the pointers `sBrng` and `dBrng` to the generator implementations for single and double precision values, respectively; and pass the pointer `InitStream` to the stream initialization routine. See recommendations below on defining such functions with input and output arguments. An example of the registration procedure for a user-designed generator can be found in the respective directory of VS examples.

The respective pointers are defined as follows:

```c
typedef int(*InitStreamPtr)( int method, VSLStreamStatePtr stream, int n, const unsigned int params[] );
typedef int(*sBRngPtr)( VSLStreamStatePtr stream, int n, float r[], float a, float b );
typedef int(*dBRngPtr)( VSLStreamStatePtr stream, int n, double r[], double a, double b );
typedef int(*iBRngPtr)( VSLStreamStatePtr stream, int n, unsigned int r[] );
```

**InitStream**

```c
int MyBrngInitStream( int method, VSLStreamStatePtr stream, int n, const unsigned int params[] )
{
  /* Initialize the stream */
  ...
} /* MyBrngInitStream */
```

**Description**

The initialization routine of a user-designed generator must initialize `stream` according to the specified initialization `method`, initial conditions `params` and the argument `n`. The value of `method` determines the initialization method to be used.

- **If `method` is equal to 1**, the initialization is by the standard generation method, which must be supported by all basic generators. In this case the function assumes that the `stream` structure was not previously initialized. The value of `n` is used as the actual number of 32-bit values passed as initial conditions through `params`. Note, that the situation when the actual number of initial conditions passed to the function is not sufficient to initialize the generator is not an error. Whenever it occurs, the basic generator must initialize the missing conditions using default settings.

- **If `method` is equal to 2**, the generation is by the leapfrog method, where `n` specifies the number of computational nodes (independent streams). Here the function assumes that the `stream` was previously initialized by the standard generation method. In this case `params` contains only one element, which identifies the computational node. If the generator does not support the leapfrog method, the function must return the error code `VSL_RNG_ERROR_LEAPFROG_UNSUPPORTED`.

- **If `method` is equal to 3**, the generation is by the block-splitting method. Same as above, the `stream` is assumed to be previously initialized by the standard generation method; `params` is not used, `n` identifies the number of skipped elements. If the generator does not support the block-splitting method, the function must return the error code `VSL_RNG_ERROR_SKIPAHEAD_UNSUPPORTED`.

- **If `method` is equal to 4**, the generation is by the advanced block-splitting method. The stream is assumed to be previously initialized by the standard generation method; `params` is converted to `MKL_UINT64[]` and `n` is used as actual number of 64-bit values in `params`. If the generator does not support the advanced block-splitting method, the function must return the error code `VSL_RNG_ERROR_SKIPAHEADEX_UNSUPPORTED`.

For a more detailed description of the leapfrog and the block-splitting methods, refer to the description of `vslLeapfrogStream`, `vslSkipAheadStream`, and `vslSkipAheadStreamEx`, respectively.
Stream state structure is individual for every generator. However, each structure has a number of fields that are the same for all the generators:

```c
typedef struct 
{
    unsigned int Reserved1[2];
    unsigned int Reserved2[2];
    [fields specific for the given generator]
} MyStreamState;
```

The fields `Reserved1` and `Reserved2` are reserved for private needs only, and must not be modified by the user. When including specific fields into the structure, follow the rules below:

- The fields must fully describe the current state of the generator. For example, the state of a linear congruential generator can be identified by only one initial condition;
- If the generator can use both the leapfrog and the block-splitting methods, additional fields should be introduced to identify the independent streams. For example, in \( LCG(a, c, m) \), apart from the initial conditions, two more fields should be specified: the value of the multiplier \( a^k \) and the value of the increment \( (a^k-1)c/(a-1) \).

For a more detailed discussion, refer to [Knuth81], and [Gentle98]. An example of the registration procedure can be found in the respective directory of VS examples.

**iBRng**

```c
int iMyBrng( VSLStreamStatePtr stream, int n, unsigned int r[] )
{
    int i; /* Loop variable */
    /* Generating integer random numbers */
    /* Pay attention to word size needed to store only random number */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* iMyBrng */
```

**NOTE**

When using 64 and 128-bit generators, consider digit capacity to store the numbers to the random vector \( r \) correctly. For example, storing one 64-bit value requires two elements of \( r \), the first to store the lower 32 bits and the second to store the higher 32 bits. Similarly, use 4 elements of \( r \) to store a 128-bit value.

**sBRng**

```c
int sMyBrng( VSLStreamStatePtr stream, int n, float r[], float a, float b )
{
    int i; /* Loop variable */
    /* Generating float (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
```
... return errcode;
} /* sMyBrng */

**dBRng**

```c
int dMyBrng( VSLStreamStatePtr stream, int n, double r[], double a, double b )
{
    int i;   /* Loop variable */
    /* Generating double (a,b) random numbers */
    for ( i = 0; i < n; i++ )
    {
        r[i] = ...;
    }
    /* Update stream state */
    ...
    return errcode;
} /* dMyBrng */
```

### Convolution and Correlation

Intel® MKL VS provides a set of routines intended to perform linear convolution and correlation transformations for single and double precision real and complex data.

For correct definition of implemented operations, see the Mathematical Notation and Definitions section.

The current implementation provides:

- Fourier algorithms for one-dimensional single and double precision real and complex data
- Fourier algorithms for multi-dimensional single and double precision real and complex data
- Direct algorithms for one-dimensional single and double precision real and complex data
- Direct algorithms for multi-dimensional single and double precision real and complex data

One-dimensional algorithms cover the following functions from the IBM® ESSL library:

- `SCONF`, `SCORF`
- `SCOND`, `SCORD`
- `SDCON`, `SDCOR`
- `DDCON`, `DDCOR`
- `SDDCON`, `SDDCOR`.

Special wrappers are designed to simulate these ESSL functions. The wrappers are provided as sample sources:

```
${MKL}/examples/vslc/essl/vsl_wrappers
```

Additionally, you can browse the examples demonstrating the calculation of the ESSL functions through the wrappers:

```
${MKL}/examples/vslc/essl
```

The convolution and correlation API provides interfaces for Fortran 90 and C/89 languages. You can use the C89 interface with later versions of the C/C++.

Intel® MKL provides the `mkl_vsl.h` header file. All header files are in the directory

```
${MKL}/include
```
The convolution and correlation API is implemented through task objects, or tasks. Task object is a data structure, or descriptor, which holds parameters that determine the specific convolution or correlation operation. Such parameters may be precision, type, and number of dimensions of user data, an identifier of the computation algorithm to be used, shapes of data arrays, and so on.

All the Intel® MKL VS convolution and correlation routines process task objects in one way or another: either create a new task descriptor, change the parameter settings, compute mathematical results of the convolution or correlation using the stored parameters, or perform other operations. Accordingly, all routines are split into the following groups:

**Task Constructors** - routines that create a new task object descriptor and set up most common parameters.

**Task Editors** - routines that can set or modify some parameter settings in the existing task descriptor.

**Task Execution Routines** - compute results of the convolution or correlation operation over the actual input data, using the operation parameters held in the task descriptor.

**Task Copy** - routines used to make several copies of the task descriptor.

**Task Destructors** - routines that delete task objects and free the memory.

When the task is executed or copied for the first time, a special process runs which is called task commitment. During this process, consistency of task parameters is checked and the required work data are prepared. If the parameters are consistent, the task is tagged as committed successfully. The task remains committed until you edit its parameters. Hence, the task can be executed multiple times after a single commitment process. Since the task commitment process may include costly intermediate calculations such as preparation of Fourier transform of input data, launching the process only once can help speed up overall performance.

**Convolution and Correlation Naming Conventions**

The names of routines, types, and constants in the convolution and correlation API are case-sensitive and can contain both lowercase and uppercase characters (`vslsConvExec`).

The names of routines have the following structure:

`vsl[datatype]{Conv|Corr}<base name>`

where

- `vsl` is a prefix indicating that the routine belongs to Intel® MKL Vector Statistics.
- `[datatype]` is optional. If present, the symbol specifies the type of the input and output data and can be `s` (for single precision real type), `d` (for double precision real type), `c` (for single precision complex type), or `z` (for double precision complex type).
- Conv or Corr specifies whether the routine refers to convolution or correlation task, respectively.
- `<base name>` field specifies a particular functionality that the routine is designed for, for example, `NewTask`, `DeleteTask`.

**Convolution and Correlation Data Types**

All convolution or correlation routines use the following types for specifying data objects:

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSLConvTaskPtr</td>
<td>Pointer to a task descriptor for convolution</td>
</tr>
<tr>
<td>VSLCorrTaskPtr</td>
<td>Pointer to a task descriptor for correlation</td>
</tr>
<tr>
<td>float</td>
<td>Input/output user real data in single precision</td>
</tr>
<tr>
<td>double</td>
<td>Input/output user real data in double precision</td>
</tr>
<tr>
<td>MKL_Complex8</td>
<td>Input/output user complex data in single precision</td>
</tr>
</tbody>
</table>
**Type** | **Data Object**
--- | ---
MKL_Complex16 | Input/output user complex data in double precision
int | All other data

Generic integer type (without specifying the byte size) is used for all integer data.

**NOTE**
The actual size of the generic integer type is platform-dependent. Before you compile your application, set an appropriate byte size for integers. See details in the 'Using the ILP64 Interface vs. LP64 Interface' section of the Intel® MKL Developer Guide.

### Convolution and Correlation Parameters

Basic parameters held by the task descriptor are assigned values when the task object is created, copied, or modified by task editors. Parameters of the correlation or convolution task are initially set up by task constructors when the task object is created. Parameter changes or additional settings are made by task editors. More parameters which define location of the data being convolved need to be specified when the task execution routine is invoked.

According to how the parameters are passed or assigned values, all of them can be categorized as either explicit (directly passed as routine parameters when a task object is created or executed) or optional (assigned some default or implicit values during task construction).

The following table lists all applicable parameters used in the Intel® MKL convolution and correlation API.

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>job</td>
<td>explicit</td>
<td>integer</td>
<td>Implied by the constructor name</td>
<td>Specifies whether the task relates to convolution or correlation</td>
</tr>
<tr>
<td>type</td>
<td>explicit</td>
<td>integer</td>
<td>Implied by the constructor name</td>
<td>Specifies the type (real or complex) of the input/output data. Set to real in the current version.</td>
</tr>
<tr>
<td>precision</td>
<td>explicit</td>
<td>integer</td>
<td>Implied by the constructor name</td>
<td>Specifies precision (single or double) of the input/output data to be provided in arrays x,y,z.</td>
</tr>
<tr>
<td>mode</td>
<td>explicit</td>
<td>integer</td>
<td>None</td>
<td>Specifies whether the convolution/correlation computation should be done via Fourier transforms, or by a direct method, or by automatically choosing between the two. See <code>SetMode</code> for the list of named constants for this parameter.</td>
</tr>
<tr>
<td>method</td>
<td>optional</td>
<td>integer</td>
<td>&quot;auto&quot;</td>
<td>Hints at a particular computation method if several methods are available for the given <code>mode</code>. Setting this parameter to &quot;auto&quot; means that software will choose the best available method.</td>
</tr>
<tr>
<td>internal_precision</td>
<td>optional</td>
<td>integer</td>
<td>Set equal to the value of <code>precision</code></td>
<td>Specifies precision of internal calculations. Can enforce double precision calculations even when input/output data are single-precision.</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
<th>Type</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dims</td>
<td>explicit</td>
<td>integer</td>
<td>None</td>
<td>Specifies the rank (number of dimensions) of the user data provided in arrays (x, y, z). Can be in the range from 1 to 7.</td>
</tr>
<tr>
<td>(x, y)</td>
<td>explicit</td>
<td>real arrays</td>
<td>None</td>
<td>Specify input data arrays. See Data Allocation for more information.</td>
</tr>
<tr>
<td>(z)</td>
<td>explicit</td>
<td>real array</td>
<td>None</td>
<td>Specifies output data array. See Data Allocation for more information.</td>
</tr>
<tr>
<td>xshape, yshape, zshape</td>
<td>explicit</td>
<td>integer arrays</td>
<td>None</td>
<td>Define shapes of the arrays (x, y, z). See Data Allocation for more information.</td>
</tr>
<tr>
<td>xstride, ystride, zstride</td>
<td>explicit</td>
<td>integer arrays</td>
<td>None</td>
<td>Define strides within arrays (x, y, z), that is specify the physical location of the input and output data in these arrays. See Data Allocation for more information.</td>
</tr>
<tr>
<td>start</td>
<td>optional</td>
<td>integer array</td>
<td>Undefined</td>
<td>Defines the first element of the mathematical result that will be stored to output array (z). See SetStart and Data Allocation for more information.</td>
</tr>
<tr>
<td>decimation</td>
<td>optional</td>
<td>integer array</td>
<td>Undefined</td>
<td>Defines how to thin out the mathematical result that will be stored to output array (z). See SetDecimation and Data Allocation for more information.</td>
</tr>
</tbody>
</table>

Users may pass the NULL pointer instead of either or all of the parameters \(x\text{stride}, y\text{stride}, \text{or } z\text{stride}\) for multi-dimensional calculations. In this case, the software assumes the dense data allocation for the arrays \(x, y, \text{or } z\) due to the Fortran-style "by columns" representation of multi-dimensional arrays.

### Convolution and Correlation Task Status and Error Reporting

The task status is an integer value, which is zero if no error has been detected while processing the task, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings.

An error can be caused by invalid parameter values, a system fault like a memory allocation failure, or can be an internal error self-detected by the software.

Each task descriptor contains the current status of the task. When creating a task object, the constructor assigns the VSL_STATUS_OK status to the task. When processing the task afterwards, other routines such as editors or executors can change the task status if an error occurs and write a corresponding error code into the task status field.

Note that at the stage of creating a task or editing its parameters, the set of parameters may be inconsistent. The parameter consistency check is only performed during the task commitment operation, which is implicitly invoked before task execution or task copying. If an error is detected at this stage, task execution or task copying is terminated and the task descriptor saves the corresponding error code. Once an error occurs, any further attempts to process that task descriptor is terminated and the task keeps the same error code.

Normally, every convolution or correlation function (except DeleteTask) returns the status assigned to the task while performing the function operation.
The header files define symbolic names for the status codes. These names are defined as macros via the `#define` statements.

If there is no error, the `VSL_STATUS_OK` status is returned, which is defined as zero:

```c
#define VSL_STATUS_OK 0
```

In case of an error, a non-zero error code is returned, which indicates the origin of the failure. The following status codes for the convolution/correlation error codes are pre-defined in the header files.

### Convolution/Correlation Status Codes

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_CC_ERROR_NOT_IMPLEMENTED</td>
<td>Requested functionality is not implemented.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_ALLOCATION_FAILURE</td>
<td>Memory allocation failure.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_BAD_DESCRIPTOR</td>
<td>Task descriptor is corrupted.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_SERVICE_FAILURE</td>
<td>A service function has failed.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_EDIT_FAILURE</td>
<td>Failure while editing the task.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_EDIT_PROHIBITED</td>
<td>You cannot edit this parameter.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_COMMIT_FAILURE</td>
<td>Task commitment has failed.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_COPY_FAILURE</td>
<td>Failure while copying the task.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_DELETE_FAILURE</td>
<td>Failure while deleting the task.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_BAD_ARGUMENT</td>
<td>Bad argument or task parameter.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_JOB</td>
<td>Bad parameter: <code>job</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_KIND</td>
<td>Bad parameter: <code>kind</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_MODE</td>
<td>Bad parameter: <code>mode</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_METHOD</td>
<td>Bad parameter: <code>method</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_TYPE</td>
<td>Bad parameter: <code>type</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_EXTERNAL_PRECISION</td>
<td>Bad parameter: <code>external_precision</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_INTERNAL_PRECISION</td>
<td>Bad parameter: <code>internal_precision</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_PRECISION</td>
<td>Incompatible external/internal precisions.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_DIMS</td>
<td>Bad parameter: <code>dims</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_XSHAPE</td>
<td>Bad parameter: <code>xshape</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_YSHAPE</td>
<td>Bad parameter: <code>yshape</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_ZSHAPE</td>
<td>Bad parameter: <code>zshape</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_XSTRIDE</td>
<td>Bad parameter: <code>xstride</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_YSTRIDE</td>
<td>Bad parameter: <code>ystride</code>.</td>
</tr>
<tr>
<td>VSL_CC_ERROR_ZSTRIDE</td>
<td>Bad parameter: <code>zstride</code>.</td>
</tr>
</tbody>
</table>

Callback function for an abstract BRNG returns an invalid number of updated entries in a buffer, that is, `< 0` or `> nmax`.

### Statistical Functions

82115
**Convolution and Correlation Task Constructors**

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters. No additional parameter adjustment is typically required and other routines can use the task object.

Intel® MKL implementation of the convolution and correlation API provides two different forms of constructors: a general form and an X-form. X-form constructors work in the same way as the general form constructors but also assign particular data to the first operand vector used in the convolution or correlation operation (stored in array $x$).

Using X-form constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector held in array $x$ against different vectors held in array $y$. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

Each constructor routine has an associated one-dimensional version that provides algorithmic and computational benefits.

**NOTE**

If the constructor fails to create a task descriptor, it returns the NULL task pointer.

The Table "Task Constructors" lists available task constructors:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vslConvNewTask/vslCorrNewTask</td>
<td>Creates a new convolution or correlation task descriptor for a multidimensional case.</td>
</tr>
<tr>
<td>vslConvNewTask1D/vslCorrNewTask1D</td>
<td>Creates a new convolution or correlation task descriptor for a one-dimensional case.</td>
</tr>
<tr>
<td>vslConvNewTaskX/vslCorrNewTaskX</td>
<td>Creates a new convolution or correlation task descriptor as an X-form for a multidimensional case.</td>
</tr>
<tr>
<td>vslConvNewTaskX1D/vslCorrNewTaskX1D</td>
<td>Creates a new convolution or correlation task descriptor as an X-form for a one-dimensional case.</td>
</tr>
</tbody>
</table>

### vslConvNewTask/vslCorrNewTask

*Creates a new convolution or correlation task descriptor for multidimensional case.*

**Syntax**

```c
status = vsisConvNewTask(task, mode, dims, xshape, yshape, zshape);
```
status = vsldConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzConvNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslsCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vsldCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslcCorrNewTask(task, mode, dims, xshape, yshape, zshape);
status = vslzCorrNewTask(task, mode, dims, xshape, yshape, zshape);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>const MKL_INT</td>
<td>Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table &quot;Values of mode parameter&quot; for a list of possible values.</td>
</tr>
<tr>
<td>dims</td>
<td>const MKL_INT</td>
<td>Rank of user data. Specifies number of dimensions for the input and output arrays x, y, and z used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.</td>
</tr>
<tr>
<td>xshape</td>
<td>const int[]</td>
<td>Defines the shape of the input data for the source array x. See Data Allocation for more information.</td>
</tr>
<tr>
<td>yshape</td>
<td>const int[]</td>
<td>Defines the shape of the input data for the source array y. See Data Allocation for more information.</td>
</tr>
<tr>
<td>zshape</td>
<td>const int[]</td>
<td>Defines the shape of the output data to be stored in array z. See Data Allocation for more information.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.</td>
</tr>
</tbody>
</table>
Description

Each vslConvNewTask/vslCorrNewTask constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters").

The parameters xshape, yshape, and zshape define the shapes of the input and output data provided by the arrays x, y, and z, respectively. Each shape parameter is an array of integers with its length equal to the value of dims. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter dims is 1, then xshape, yshape, zshape are equal to the number of elements read from the arrays x and y or stored to the array z. Note that values of shape parameters may differ from physical shapes of arrays x, y, and z if non-trivial strides are assigned.

If the constructor fails to create a task descriptor, it returns a NULL task pointer.

vslConvNewTask1D/vslCorrNewTask1D

Creates a new convolution or correlation task descriptor for one-dimensional case.

Syntax

status = vslsConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzConvNewTask1D(task, mode, xshape, yshape, zshape);
status = vslsCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslcCorrNewTask1D(task, mode, xshape, yshape, zshape);
status = vslzCorrNewTask1D(task, mode, xshape, yshape, zshape);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>const MKL_INT</td>
<td>Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table &quot;Values of mode parameter&quot; for a list of possible values.</td>
</tr>
<tr>
<td>xshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the input data sequence for the source array x. See Data Allocation for more information.</td>
</tr>
<tr>
<td>yshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the input data sequence for the source array y. See Data Allocation for more information.</td>
</tr>
<tr>
<td>zshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the output data sequence to be stored in array z. See Data Allocation for more information.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr* for vsslConvNewTask1D, vsldConvNewTask1D, vslcConvNewTask1D, vslzConvNewTask1D</td>
<td>Pointer to the task descriptor if created successfully or NULL pointer otherwise.</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.</td>
</tr>
</tbody>
</table>

Description

Each vslConvNewTask1D/vslCorrNewTask1D constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters"). Unlike vslConvNewTask/vslCorrNewTask, these routines represent a special one-dimensional version of the constructor which assumes that the value of the parameter dims is 1. The parameters xshape, yshape, and zshape are equal to the number of elements read from the arrays x and y or stored to the array z. You explicitly assign the shape parameters when calling the constructor.

vslConvNewTaskX/vslCorrNewTaskX

Creates a new convolution or correlation task descriptor for multidimensional case and assigns source data to the first operand vector.

Syntax

```
status = vsslConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzConvNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsslCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslcCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
status = vslzCorrNewTaskX(task, mode, dims, xshape, yshape, zshape, x, xstride);
```

Include Files

- mkl.h
## Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>const MKL_INT</td>
<td>Specifies whether convolution/correlation calculation must be performed by using a direct algorithm or through Fourier transform of the input data. See Table &quot;Values of mode parameter&quot; for a list of possible values.</td>
</tr>
<tr>
<td>dims</td>
<td>const MKL_INT</td>
<td>Rank of user data. Specifies number of dimensions for the input and output arrays x, y, and z used during the execution stage. Must be in the range from 1 to 7. The value is explicitly assigned by the constructor.</td>
</tr>
<tr>
<td>xshape</td>
<td>const int[]</td>
<td>Defines the shape of the input data for the source array x. See Data Allocation for more information.</td>
</tr>
<tr>
<td>yshape</td>
<td>const int[]</td>
<td>Defines the shape of the input data for the source array y. See Data Allocation for more information.</td>
</tr>
<tr>
<td>zshape</td>
<td>const int[]</td>
<td>Defines the shape of the output data to be stored in array z. See Data Allocation for more information.</td>
</tr>
<tr>
<td>x</td>
<td>const float[]</td>
<td>Pointer to the array containing input data for the first operand vector. See Data Allocation for more information.</td>
</tr>
<tr>
<td>x</td>
<td>const double[]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>const MKL_Complex8[]</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>const MKL_Complex16[]</td>
<td></td>
</tr>
<tr>
<td>xstride</td>
<td>const int[]</td>
<td>Strides for input data in the array x.</td>
</tr>
</tbody>
</table>

## Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr*</td>
<td>Pointer to the task descriptor if created successfully or NULL pointer otherwise.</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is created successfully or set to non-zero error code otherwise.</td>
</tr>
</tbody>
</table>
Description
Each `vslConvNewTaskX/vslCorrNewTaskX` constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table “Convolution and Correlation Task Parameters”).

Unlike `vslConvNewTask/vslCorrNewTask`, these routines represent the so called X-form version of the constructor, which means that in addition to creating the task descriptor they assign particular data to the first operand vector in array \( x \) used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX/vslCorrNewTaskX` constructor keeps the pointer to the array \( x \) all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array \( x \) against different vectors in array \( y \). This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` define the shapes of the input and output data provided by the arrays \( x \), \( y \), and \( z \), respectively. Each shape parameter is an array of integers with its length equal to the value of `dims`. You explicitly assign the shape parameters when calling the constructor. If the value of the parameter `dims` is 1, then `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays \( x \) and \( y \) or stored to the array \( z \). Note that values of shape parameters may differ from physical shapes of arrays \( x \), \( y \), and \( z \) if non-trivial strides are assigned.

The stride parameter `xstride` specifies the physical location of the input data in the array \( x \). In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is \( s \), then only every \( s \)th element of the array \( x \) will be used to form the input sequence. The stride value must be positive or negative but not zero.

**vslConvNewTaskX1D/vslCorrNewTaskX1D**

*Creates a new convolution or correlation task descriptor for one-dimensional case and assigns source data to the first operand vector.*

**Syntax**

```c
status = vslsConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslcConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslzConvNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslsCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vsldCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslcCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
status = vslzCorrNewTaskX1D(task, mode, xshape, yshape, zshape, x, xstride);
```

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>const MKL_INT</td>
<td>Specifies whether convolution/correlation calculation must be performed by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>using a direct algorithm or through Fourier transform of the input data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Table &quot;Values of mode parameter&quot; for a list of possible values.</td>
</tr>
<tr>
<td>xshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the input data sequence for the source array x.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Data Allocation for more information.</td>
</tr>
<tr>
<td>yshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the input data sequence for the source array y.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Data Allocation for more information.</td>
</tr>
<tr>
<td>zshape</td>
<td>const MKL_INT</td>
<td>Defines the length of the output data sequence to be stored in array z.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See Data Allocation for more information.</td>
</tr>
<tr>
<td>x</td>
<td>const float[] for real</td>
<td>Pointer to the array containing input data for the first operand vector.</td>
</tr>
<tr>
<td></td>
<td>data in single precision</td>
<td>See Data Allocation for more information.</td>
</tr>
<tr>
<td></td>
<td>flavors,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double[] for real</td>
<td></td>
</tr>
<tr>
<td></td>
<td>data in double precision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flavors,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex8[] for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complex data in single</td>
<td></td>
</tr>
<tr>
<td></td>
<td>precision flavors,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_Complex16[] for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>complex data in double</td>
<td></td>
</tr>
<tr>
<td></td>
<td>precision flavors</td>
<td></td>
</tr>
<tr>
<td>xstride</td>
<td>const MKL_INT</td>
<td>Stride for input data sequence in the array x.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr* for</td>
<td>Pointer to the task descriptor if created successfully or NULL pointer</td>
</tr>
<tr>
<td></td>
<td>vslsConvNewTaskX1D,</td>
<td>otherwise.</td>
</tr>
<tr>
<td></td>
<td>vsldConvNewTaskX1D,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslcConvNewTaskX1D,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslzConvNewTaskX1D</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VSLCorrTaskPtr* for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslsCorrNewTaskX1D,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vsldCorrNewTaskX1D,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslcCorrNewTaskX1D,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslzCorrNewTaskX1D</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is created successfully or set to non-zero</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error code otherwise.</td>
</tr>
</tbody>
</table>

Description

Each vslConvNewTaskX1D/vslCorrNewTaskX1D constructor creates a new convolution or correlation task descriptor with the user specified values for explicit parameters. The optional parameters are set to their default values (see Table "Convolution and Correlation Task Parameters").
These routines represent a special one-dimensional version of the so called X-form of the constructor. This assumes that the value of the parameter `dims` is 1 and that in addition to creating the task descriptor, constructor routines assign particular data to the first operand vector in array `x` used in convolution or correlation operation. The task descriptor created by the `vslConvNewTaskX1D/vslCorrNewTaskX1D` constructor keeps the pointer to the array `x` all the time, that is, until the task object is deleted by one of the destructor routines (see `vslConvDeleteTask/vslCorrDeleteTask`).

Using this form of constructors is recommended when you need to compute multiple convolutions or correlations with the same data vector in array `x` against different vectors in array `y`. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

The parameters `xshape`, `yshape`, and `zshape` are equal to the number of elements read from the arrays `x` and `y` or stored to the array `z`. You explicitly assign the shape parameters when calling the constructor.

The stride parameters `xstride` specifies the physical location of the input data in the array `x` and is an interval between locations of consecutive elements of the array. For example, if the value of the parameter `xstride` is `s`, then only every `s`th element of the array `x` will be used to form the input sequence. The stride value must be positive or negative but not zero.

### Convolution and Correlation Task Editors

Task editors in convolution and correlation API of Intel® MKL are routines intended for setting up or changing the following task parameters (see Table "Convolution and Correlation Task Parameters"):

- `mode`
- `internal_precision`
- `start`
- `decimation`

For setting up or changing each of the above parameters, a separate routine exists.

**NOTE**

Fields of the task descriptor structure are accessible only through the set of task editor routines provided with the software.

The work data computed during the last commitment process may become invalid with respect to new parameter settings. That is why after applying any of the editor routines to change the task descriptor settings, the task loses its commitment status and goes through the full commitment process again during the next execution or copy operation. For more information on task commitment, see the Introduction to Convolution and Correlation.

Table "Task Editors" lists available task editors.

### Task Editors

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vslConvSetMode</code>/<code>vslCorrSetMode</code></td>
<td>Changes the value of the parameter <code>mode</code> for the operation of convolution or correlation.</td>
</tr>
<tr>
<td><code>vslConvSetInternalPrecision</code>/<code>vslCorrSetInternalPrecision</code></td>
<td>Changes the value of the parameter <code>internal_precision</code> for the operation of convolution or correlation.</td>
</tr>
<tr>
<td><code>vslConvSetStart</code>/<code>vslCorrSetStart</code></td>
<td>Sets the value of the parameter <code>start</code> for the operation of convolution or correlation.</td>
</tr>
</tbody>
</table>
### Routine Description

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vslConvSetDecimation</code>/<code>vslCorrSetDecimation</code></td>
<td>Sets the value of the parameter <code>decimation</code> for the operation of convolution or correlation.</td>
</tr>
</tbody>
</table>

**NOTE**
You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

---

### `vslConvSetMode/vslCorrSetMode`

*Changes the value of the parameter `mode` in the convolution or correlation task descriptor.*

**Syntax**

```c
status = vslConvSetMode(task, newmode);
status = vslCorrSetMode(task, newmode);
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td><code>VSLConvTaskPtr</code> for <code>vslConvSetMode</code></td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td></td>
<td><code>VSLCorrTaskPtr</code> for <code>vslCorrSetMode</code></td>
<td></td>
</tr>
<tr>
<td>newmode</td>
<td><code>const MKL_INT</code></td>
<td>New value of the parameter <code>mode</code>.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td><code>int</code></td>
<td>Current status of the task.</td>
</tr>
</tbody>
</table>

**Description**

This function is declared in `mkl_vsl_functions.h`.

The function routine changes the value of the parameter `mode` for the operation of convolution or correlation. This parameter defines whether the computation should be done via Fourier transforms of the input/output data or using a direct algorithm. Initial value for `mode` is assigned by a task constructor.

Predefined values for the `mode` parameter are as follows:

**Values of `mode` parameter**

<table>
<thead>
<tr>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>VSL_CONV_MODE_FFT</code></td>
<td>Compute convolution by using fast Fourier transform.</td>
</tr>
<tr>
<td>Value</td>
<td>Purpose</td>
</tr>
<tr>
<td>--------------------</td>
<td>----------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_CORR_MODE_FFT</td>
<td>Compute correlation by using fast Fourier transform.</td>
</tr>
<tr>
<td>VSL_CONV_MODE_DIRECT</td>
<td>Compute convolution directly.</td>
</tr>
<tr>
<td>VSL_CORR_MODE_DIRECT</td>
<td>Compute correlation directly.</td>
</tr>
<tr>
<td>VSL_CONV_MODE_AUTO</td>
<td>Automatically choose direct or Fourier mode for convolution.</td>
</tr>
<tr>
<td>VSL_CORR_MODE_AUTO</td>
<td>Automatically choose direct or Fourier mode for correlation.</td>
</tr>
</tbody>
</table>

**vslConvSetInternalPrecision/vslCorrSetInternalPrecision**

Changes the value of the parameter `internal_precision` in the convolution or correlation task descriptor.

**Syntax**

```c
status = vslConvSetInternalPrecision(task, precision);
status = vslCorrSetInternalPrecision(task, precision);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr for</td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td></td>
<td>vslConvSetInternalPrecision</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VSLCorrTaskPtr for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vslCorrSetInternalPrecision</td>
<td></td>
</tr>
<tr>
<td>precision</td>
<td>const MKL_INT</td>
<td>New value of the parameter <code>internal_precision</code>.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task.</td>
</tr>
</tbody>
</table>

**Description**

The `vslConvSetInternalPrecision/vslCorrSetInternalPrecision` routine changes the value of the parameter `internal_precision` for the operation of convolution or correlation. This parameter defines whether the internal computations of the convolution or correlation result should be done in single or double precision. Initial value for `internal_precision` is assigned by a task constructor and set to either "single" or "double" according to the particular flavor of the constructor used.

Changing the `internal_precision` can be useful if the default setting of this parameter was "single" but you want to calculate the result with double precision even if input and output data are represented in single precision.

Predefined values for the `internal_precision` input parameter are as follows:
Values of *internal_precision* Parameter

<table>
<thead>
<tr>
<th>Value</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_CONV_PRECISION_SINGLE</td>
<td>Compute convolution with single precision.</td>
</tr>
<tr>
<td>VSL_CORR_PRECISION_SINGLE</td>
<td>Compute correlation with single precision.</td>
</tr>
<tr>
<td>VSL_CONV_PRECISION_DOUBLE</td>
<td>Compute convolution with double precision.</td>
</tr>
<tr>
<td>VSL_CORR_PRECISION_DOUBLE</td>
<td>Compute correlation with double precision.</td>
</tr>
</tbody>
</table>

**vslConvSetStart/vslCorrSetStart**

*Changes the value of the parameter start in the convolution or correlation task descriptor.*

**Syntax**

```c
status = vslConvSetStart(task, start);
status = vslCorrSetStart(task, start);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr for vslConvSetStart</td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td>start</td>
<td>const int[]</td>
<td>New value of the parameter \textit{start}.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task.</td>
</tr>
</tbody>
</table>

**Description**

The `vslConvSetStart/vslCorrSetStart` routine sets the value of the parameter \textit{start} for the operation of convolution or correlation. In a one-dimensional case, this parameter points to the first element in the mathematical result that should be stored in the output array. In a multidimensional case, \textit{start} is an array of indices and its length is equal to the number of dimensions specified by the parameter \textit{dims}. For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for \textit{start} is undefined and this parameter is not used. Therefore the only way to set and use the \textit{start} parameter is via assigning it some value by one of the `vslConvSetStart/vslCorrSetStart` routines.
vslConvSetDecimation/vslCorrSetDecimation
Changes the value of the parameter decimation in the convolution or correlation task descriptor.

Syntax

\[
\text{status} = \text{vslConvSetDecimation}(\text{task}, \text{decimation}); \\
\text{status} = \text{vslCorrSetDecimation}(\text{task}, \text{decimation});
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr for vslConvSetDecimation VSLCorrTaskPtr for vslCorrSetDecimation</td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td>decimation</td>
<td>const int[]</td>
<td>New value of the parameter decimation.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task.</td>
</tr>
</tbody>
</table>

Description

The routine sets the value of the parameter decimation for the operation of convolution or correlation. This parameter determines how to thin out the mathematical result of convolution or correlation before writing it into the output data array. For example, in a one-dimensional case, if \( \text{decimation} = d > 1 \), only every \( d \)-th element of the mathematical result is written to the output array \( z \). In a multidimensional case, \( \text{decimation} \) is an array of indices and its length is equal to the number of dimensions specified by the parameter \( \text{dims} \). For more information about the definition and effect of this parameter, see Data Allocation.

During the initial task descriptor construction, the default value for \( \text{decimation} \) is undefined and this parameter is not used. Therefore the only way to set and use the \( \text{decimation} \) parameter is via assigning it some value by one of the vslSetDecimation routines.

Task Execution Routines

Task execution routines compute convolution or correlation results based on parameters held by the task descriptor and on the user data supplied for input vectors.

After you create and adjust a task, you can execute it multiple times by applying to different input/output data of the same type, precision, and shape.

Intel® MKL provides the following forms of convolution/correlation execution routines:

- **General form** executors that use the task descriptor created by the general form constructor and expect to get two source data arrays \( x \) and \( y \) on input
• **X-form** executors that use the task descriptor created by the X-form constructor and expect to get only one source data array \( y \) on input because the first array \( x \) has been already specified on the construction stage.

When the task is executed for the first time, the execution routine includes a task commitment operation, which involves two basic steps: parameters consistency check and preparation of auxiliary data (for example, this might be the calculation of Fourier transform for input data).

Each execution routine has an associated one-dimensional version that provides algorithmic and computational benefits.

**NOTE**
You can use the `NULL` task pointer in calls to execution routines. In this case, the routine is terminated and no system crash occurs.

If the task is executed successfully, the execution routine returns the zero status code. If an error is detected, the execution routine returns an error code which signals that a specific error has occurred. In particular, an error status code is returned in the following cases:

- if the task pointer is `NULL`
- if the task descriptor is corrupted
- if calculation has failed for some other reason.

**NOTE**
Intel® MKL does not control floating-point errors, like overflow or gradual underflow, or operations with NaNs, etc.

If an error occurs, the task descriptor stores the error code.

The table below lists all task execution routines.

### Task Execution Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vslConvExec/vslCorrExec</code></td>
<td>Computes convolution or correlation for a multidimensional case.</td>
</tr>
<tr>
<td><code>vslConvExec1D/vslCorrExec1D</code></td>
<td>Computes convolution or correlation for a one-dimensional case.</td>
</tr>
<tr>
<td><code>vslConvExecX/vslCorrExecX</code></td>
<td>Computes convolution or correlation as X-form for a multidimensional case.</td>
</tr>
<tr>
<td><code>vslConvExecX1D/vslCorrExecX1D</code></td>
<td>Computes convolution or correlation as X-form for a one-dimensional case.</td>
</tr>
</tbody>
</table>

### `vslConvExec/vslCorrExec`

**Computes convolution or correlation for multidimensional case.**

**Syntax**

```c
status = vsisConvExec(task, x, xstride, y, ystride, z, zstride);
status = vsldConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslcConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslzConvExec(task, x, xstride, y, ystride, z, zstride);
status = vslsCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vsldCorrExec(task, x, xstride, y, ystride, z, zstride);
```
status = vslcCorrExec(task, x, xstride, y, ystride, z, zstride);
status = vslzCorrExec(task, x, xstride, y, ystride, z, zstride);

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr</td>
<td>Pointer to the task descriptor</td>
</tr>
</tbody>
</table>

For
- vslsConvExec, vsldConvExec, vslcConvExec, vslzConvExec

- VSLCorrTaskPtr
  - vslsCorrExec, vsldCorrExec, vslcCorrExec, vslzCorrExec

- x, y
  - const float[] for vslsConvExec and vslsCorrExec,
  - const double[] for vsldConvExec and vsldCorrExec,
  - const MKL_Complex8[] for vslcConvExec and vslcCorrExec,
  - const MKL_Complex16[] for vslzConvExec and vslzCorrExec

- xstride, ystride, zstride
  - const int[]

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>const float[] for</td>
<td>Pointer to the array that stores output data. See Data Allocation for more information.</td>
</tr>
</tbody>
</table>

For
- vslsConvExec and vslsCorrExec,
- vsldConvExec and vsldCorrExec,
- vslcConvExec and vslcCorrExec,
- vslzConvExec and vslzCorrExec,
Description

Each of the vslConvExec/vslCorrExec routines computes convolution or correlation of the data provided by the arrays \( x \) and \( y \) and then stores the results in the array \( z \). Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTask/vslCorrNewTask constructor and pointed to by \( \text{task} \). If \( \text{task} \) is NULL, no operation is done.

The stride parameters \( x\text{stride} \), \( y\text{stride} \), and \( z\text{stride} \) specify the physical location of the input and output data in the arrays \( x \), \( y \), and \( z \), respectively. In a one-dimensional case, stride is an interval between locations of consecutive elements of the array. For example, if the value of the parameter \( z\text{stride} \) is \( s \), then only every \( s^{th} \) element of the array \( z \) will be used to store the output data. The stride value must be positive or negative but not zero.

**vslConvExec1D/vslCorrExec1D**

*Computes convolution or correlation for one-dimensional case.*

**Syntax**

```c
status = vslsConvExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vsldConvExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vslcConvExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vslzConvExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vslsCorrExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vsldCorrExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vslcCorrExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
status = vslzCorrExec1D(task, x, x\text{stride}, y, y\text{stride}, z, z\text{stride});
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{task} )</td>
<td>VSLConvTaskPtr</td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>x, y</td>
<td>const float[] for vslsConvExec1D and vslsCorrExec1D, vsldConvExec1D and vsldCorrExec1D, const MKL_Complex8[] for vslcConvExec1D and vslcCorrExec1D, const MKL_Complex16[] for vslzConvExec1D and vslzCorrExec1D</td>
<td>Pointers to arrays containing input data. See Data Allocation for more information.</td>
</tr>
<tr>
<td>xstride, ystride, zstride</td>
<td>const MKL_INT</td>
<td>Strides for input and output data. For more information, see stride parameters.</td>
</tr>
<tr>
<td>z</td>
<td>const float[] for vslsConvExec1D and vslsCorrExec1D, vsldConvExec1D and vsldCorrExec1D, const double[] for vsldConvExec1D and vsldCorrExec1D, const MKL_Complex8[] for vslcConvExec1D and vslcCorrExec1D, const MKL_Complex16[] for vslzConvExec1D and vslzCorrExec1D</td>
<td>Pointer to the array that stores output data. See Data Allocation for more information.</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.</td>
</tr>
</tbody>
</table>

**Description**

Each of the `vslConvExec1D/vslCorrExec1D` routines computes convolution or correlation of the data provided by the arrays `x` and `y` and then stores the results in the array `z`. These routines represent a special one-dimensional version of the operation, assuming that the value of the parameter `dims` is 1. Using this version of execution routines can help speed up performance in case of one-dimensional data.
Parameters of the operation are read from the task descriptor created previously by a corresponding `vslConvNewTask1D`/`vslCorrNewTask1D` constructor and pointed to by `task`. If `task` is NULL, no operation is done.

**vslConvExecX/vslCorrExecX**

*Computes convolution or correlation for multidimensional case with the fixed first operand vector.*

**Syntax**

```c
status = vslConvExecX(task, y, ystride, z, zstride);
status = vsldConvExecX(task, y, ystride, z, zstride);
status = vslcConvExecX(task, y, ystride, z, zstride);
status = vslzConvExecX(task, y, ystride, z, zstride);
status = vslsConvExecX(task, y, ystride, z, zstride);
status = vslcConvExecX(task, y, ystride, z, zstride);
status = vslzConvExecX(task, y, ystride, z, zstride);
status = vslsCorrExecX(task, y, ystride, z, zstride);
status = vslcCorrExecX(task, y, ystride, z, zstride);
status = vslzCorrExecX(task, y, ystride, z, zstride);
status = vsldCorrExecX(task, y, ystride, z, zstride);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code>, <code>y</code></td>
<td>const float[] for vslConvExecX and vslsCorrExecX, const double[] for vsldConvExecX and vsldCorrExecX, const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX</td>
<td>Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.</td>
</tr>
</tbody>
</table>
### Name | Type | Description
--- | --- | ---
const MKL_Complex16[] | for vslzConvExecX and vslzCorrExecX | ystride, z const int[] stride

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
z | const float[] for vslsConvExecX and vslsCorrExecX, const double[] for vsldConvExecX and vsldCorrExecX, const MKL_Complex8[] for vslcConvExecX and vslcCorrExecX, const MKL_Complex16[] for vslzConvExecX and vslzCorrExecX | Pointer to the array that stores output data. See Data Allocation for more information. |

### Description

Each of the vslConvExecX/vslCorrExecX routines computes convolution or correlation of the data provided by the arrays \( x \) and \( y \) and then stores the results in the array \( z \). These routines represent a special version of the operation, which assumes that the first operand vector was set on the task construction stage and the task object keeps the pointer to the array \( x \).

Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTaskX/vslCorrNewTaskX constructor and pointed to by \( \text{task} \). If \( \text{task} \) is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple convolutions or correlations with the same data vector in array \( x \) against different vectors in array \( y \). This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

### Syntax

```c
status = vslsConvExecX1D(task, y, ystride, z, zstride);
status = vsldConvExecX1D(task, y, ystride, z, zstride);
status = vslcConvExecX1D(task, y, ystride, z, zstride);
```
status = vslzConvExecX1D(task, y, ystride, z, zstride);
status = vslsCorrExecX1D(task, y, ystride, z, zstride);
status = vslcCorrExecX1D(task, y, ystride, z, zstride);
status = vslzCorrExecX1D(task, y, ystride, z, zstride);
status = vsldCorrExecX1D(task, y, ystride, z, zstride);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr for vslsConvExecX1D, vsldConvExecX1D, vslcConvExecX1D, vslzConvExecX1D VSLCorrTaskPtr for vslsCorrExecX1D, vsldCorrExecX1D, vslcCorrExecX1D, vslzCorrExecX1D</td>
<td>Pointer to the task descriptor.</td>
</tr>
<tr>
<td>x, y</td>
<td>const float[] for vslsConvExecX1D and vslsCorrExecX1D, const double[] for vsldConvExecX1D and vsldCorrExecX1D, const MKL_Complex8[] for vslcConvExecX1D and vslcCorrExecX1D, const MKL_Complex16[] for vslzConvExecX1D and vslzCorrExecX1D</td>
<td>Pointer to array containing input data (for the second operand vector). See Data Allocation for more information.</td>
</tr>
<tr>
<td>ystride, zstride</td>
<td>const MKL_INT</td>
<td>Strides for input and output data. For more information, see stride parameters.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>const float[] for vslsConvExecX1D and vslsCorrExecX1D,</td>
<td>Pointer to the array that stores output data. See Data Allocation for more information.</td>
</tr>
</tbody>
</table>
### Name | Type | Description
--- | --- | ---
const double[] for vsldConvExecX1D and vsldCorrExecX1D,  
const MKL_Complex8[] for vslcConvExecX1D and vslcCorrExecX1D,  
const MKL_Complex16[] for vslzConvExecX1D and vslzCorrExecX1D

### Status | int | Set to VSL_STATUS_OK if the task is executed successfully or set to non-zero error code otherwise.

#### Description
Each of the vslConvExecX1D/vslCorrExecX1D routines computes convolution or correlation of one-dimensional (assuming that `dims` = 1) data provided by the arrays x and y and then stores the results in the array z. These routines represent a special version of the operation, which expects that the first operand vector was set on the task construction stage.

Parameters of the operation are read from the task descriptor created previously by a corresponding vslConvNewTaskX1D/vslCorrNewTaskX1D constructor and pointed to by `task`. If `task` is NULL, no operation is done.

Using this form of execution routines is recommended when you need to compute multiple one-dimensional convolutions or correlations with the same data vector in array x against different vectors in array y. This helps improve performance by eliminating unnecessary overhead in repeated computation of intermediate data required for the operation.

#### Convolution and Correlation Task Destructors

Task destructors are routines designed for deleting task objects and deallocating memory.

**vslConvDeleteTask/vslCorrDeleteTask**

*Destroys the task object and frees the memory.*

### Syntax

```c
errcode = vslConvDeleteTask(task);
errcode = vslCorrDeleteTask(task);
```

### Include Files

- mkl.h

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLConvTaskPtr* for vslConvDeleteTask</td>
<td>Pointer to the task descriptor.</td>
</tr>
</tbody>
</table>
### Description

The `vslConvDeleteTask/vslCorrDeleteTask` routine deletes the task descriptor object and frees any working memory and the memory allocated for the data structure. The task pointer is set to `NULL`.

Note that if the `vslConvDeleteTask/vslCorrDeleteTask` routine does not delete the task successfully, the routine returns an error code. This error code has no relation to the task status code and does not change it.

**NOTE**

You can use the `NULL` task pointer in calls to destructor routines. In this case, the routine terminates with no system crash.

### Convolution and Correlation Task Copiers

The routines are designed for copying convolution and correlation task descriptors.

**vslConvCopyTask/vslCorrCopyTask**

Copies a descriptor for convolution or correlation task.

**Syntax**

```c
status = vslConvCopyTask(newtask, srctask);
status = vslCorrCopyTask(newtask, srctask);
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srctask</td>
<td><code>const VSLConvTaskPtr</code> for <code>vslConvCopyTask</code> or <code>const VSLCorrTaskPtr</code> for <code>vslCorrCopyTask</code></td>
<td>Pointer to the source task descriptor.</td>
</tr>
</tbody>
</table>
### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newtask</td>
<td>VSLConvTaskPtr* for vslConvCopyTask</td>
<td>Pointer to the new task descriptor.</td>
</tr>
<tr>
<td></td>
<td>VSLCorrTaskPtr* for vslCorrCopyTask</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the source task.</td>
</tr>
</tbody>
</table>

### Description

If a task object `srctask` already exists, you can use an appropriate `vslConvCopyTask/vslCorrCopyTask` routine to make its copy in `newtask`. After the copy operation, both source and new task objects will become committed (see Introduction to Convolution and Correlation for information about task commitment). If the source task was not previously committed, the commitment operation for this task is implicitly invoked before copying starts. If an error occurs during source task commitment, the task stores the error code in the status field. If an error occurs during copy operation, the routine returns a `NULL` pointer instead of a reference to a new task object.

### Convolution and Correlation Usage Examples

This section demonstrates how you can use the Intel® MKL routines to perform some common convolution and correlation operations both for single-threaded and multithreaded calculations. The following two sample functions `scond1` and `sconf1` simulate the convolution and correlation functions `SCOND` and `SCONF` found in IBM ESSL* library. The functions assume single-threaded calculations and can be used with C or C++ compilers.

#### Function scond1 for Single-Threaded Calculations

```c
#include "mkl_vsl.h"

int scond1(
    float h[], int inch,
    float x[], int incx,
    float y[], int incy,
    int nh, int nx, int iy0, int ny)
{
    int status;
    VSLConvTaskPtr task;
    vslsConvNewTask1D(&task, VSL_CONV_MODE_DIRECT, nh, nx, ny);
    vslConvSetStart(task, &iy0);
    status = vslsConvExec1D(task, h, inch, x, incx, y, incy);
    vslConvDeleteTask(&task);
    return status;
}
```
Function sconf1 for Single-Threaded Calculations

```c
#include "mkl_vsl.h"

int sconf1(
    int init,
    float h[], int inc1h,
    float x[], int inc1x, int inc2x,
    float y[], int inc1y, int inc2y,
    int nh, int nx, int m, int iy0, int ny,
    void* aux1, int naux1, void* aux2, int naux2)
{
    int status;
    /* assume that aux1!=0 and naux1 is big enough */
    VSLConvTaskPtr* task = (VSLConvTaskPtr*)aux1;
    if (init != 0)
        /* initialization: */
        status = vslsConvNewTaskX1D(task, VSL_CONV_MODE_FFT, nh, nx, ny, h, inc1h);
    if (init == 0) {
        /* calculations: */
        int i;
        vslConvSetStart(*task, &iy0);
        for (i=0; i<m; i++) {
            float* xi = &x[inc2x * i];
            float* yi = &y[inc2y * i];
            /* task is implicitly committed at i==0 */
            status = vslsConvExecX1D(*task, xi, inc1x, yi, inc1y);
        }
    }
    vslConvDeleteTask(task);
    return status;
}
```
Using Multiple Threads

For functions such as `sconf1` described in the previous example, parallel calculations may be more preferable instead of cycling. If \(m>1\), you can use multiple threads for invoking the task execution against different data sequences. For such cases, use task copy routines to create \(m\) copies of the task object before the calculations stage and then run these copies with different threads. Ensure that you make all necessary parameter adjustments for the task (using Task Editors) before copying it.

The sample code in this case may look as follows:

```c
if (init == 0) {
    int i, status, ss[M];
    VSLConvTaskPtr tasks[M];
    /* assume that M is big enough */
    ...
    vslConvSetStart(*task, &iy0);
    ...
    for (i=0; i<m; i++)
        /* implicit commitment at i==0 */
        vslConvCopyTask(&tasks[i],*task);
    ...
}
```

Then, \(m\) threads may be started to execute different copies of the task:

```c
...
float* xi = &x[inc2x * i];
float* yi = &y[inc2y * i];
ss[i]=vslsConvExecX1D(tasks[i], xi,inc1x, yi,inc1y);
...
```

And finally, after all threads have finished the calculations, overall status should be collected from all task objects. The following code signals the first error found, if any:

```c
for (i=0; i<m; i++) {
    status = ss[i];
    if (status != 0) /* 0 means "OK" */
        break;
}
return status;
```

Execution routines modify the task internal state (fields of the task structure). Such modifications may conflict with each other if different threads work with the same task object simultaneously. That is why different threads must use different copies of the task.
Convolution and Correlation Mathematical Notation and Definitions

The following notation is necessary to explain the underlying mathematical definitions used in the text:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R = (-\infty, +\infty)$</td>
<td>The set of real numbers.</td>
</tr>
<tr>
<td>$\mathbb{Z} = {0, \pm 1, \pm 2, \ldots}$</td>
<td>The set of integer numbers.</td>
</tr>
<tr>
<td>$\mathbb{Z}^N = \mathbb{Z} \times \ldots \times \mathbb{Z}$</td>
<td>The set of $N$-dimensional series of integer numbers.</td>
</tr>
<tr>
<td>$p = (p_1, \ldots, p_N) \in \mathbb{Z}^N$</td>
<td>$N$-dimensional series of integers.</td>
</tr>
<tr>
<td>$u: \mathbb{Z}^N \to R$</td>
<td>Function $u$ with arguments from $\mathbb{Z}^N$ and values from $R$.</td>
</tr>
<tr>
<td>$w = u * v$</td>
<td>Function $w$ is the convolution of the functions $u, v$.</td>
</tr>
<tr>
<td>$w = u \cdot v$</td>
<td>Function $w$ is the correlation of the functions $u, v$.</td>
</tr>
</tbody>
</table>

Given series $p, q \in \mathbb{Z}^N$:
- series $r = p + q$ is defined as $r^n = p^n + q^n$ for every $n = 1, \ldots, N$
- series $r = p - q$ is defined as $r^n = p^n - q^n$ for every $n = 1, \ldots, N$
- series $r = \sup\{p, q\}$ is defined as $r^n = \max\{p^n, q^n\}$ for every $n = 1, \ldots, N$
- series $r = \inf\{p, q\}$ is defined as $r^n = \min\{p^n, q^n\}$ for every $n = 1, \ldots, N$
- inequality $p \preceq q$ means that $p^n \preceq q^n$ for every $n = 1, \ldots, N$.

A function $u(p)$ is called a finite function if there exist series $p_{\text{min}}, p_{\text{max}} \in \mathbb{Z}^N$ such that:

$u(p) \neq 0$ implies $p_{\text{min}} \preceq p \preceq p_{\text{max}}$.

Operations of convolution and correlation are only defined for finite functions.

Consider functions $u, v$ and series $p_{\text{min}}, p_{\text{max}}, q_{\text{min}}, Q_{\text{max}} \in \mathbb{Z}^N$ such that:

$u(p) \neq 0$ implies $p_{\text{min}} \preceq p \preceq p_{\text{max}}$.
$v(q) \neq 0$ implies $q_{\text{min}} \preceq q \preceq Q_{\text{max}}$.

Definitions of linear correlation and linear convolution for functions $u$ and $v$ are given below.

**Linear Convolution**

If function $w = u * v$ is the convolution of $u$ and $v$, then:

$w(r) \neq 0$ implies $R_{\text{min}} \preceq r \preceq R_{\text{max}}$,
where $R_{\text{min}} = p_{\text{min}} + Q_{\text{min}}$ and $R_{\text{max}} = p_{\text{max}} + Q_{\text{max}}$.

If $R_{\text{min}} \preceq r \preceq R_{\text{max}}$, then:
\( w(r) = \sum u(t) \cdot v(r-t) \) is the sum for all \( t \in \mathbb{Z}^N \) such that \( T^\text{min} \leq t \leq T^\text{max} \), where \( T^\text{min} = \sup\{p^\text{min}, r- Q^\text{max}\} \) and \( T^\text{max} = \inf\{p^\text{max}, r- Q^\text{min}\} \).

**Linear Correlation**

If function \( w = u \cdot v \) is the correlation of \( u \) and \( v \), then:

\[ w(r) \neq 0 \text{ implies } R^\text{min} \leq r \leq R^\text{max}, \]

where \( R^\text{min} = Q^\text{min} - p^\text{max} \) and \( R^\text{max} = Q^\text{max} - p^\text{min} \).

If \( R^\text{min} \leq r \leq R^\text{max} \), then:

\[ w(r) = \sum u(t) \cdot v(r+t) \] is the sum for all \( t \in \mathbb{Z}^N \) such that \( T^\text{min} \leq t \leq T^\text{max} \), where \( T^\text{min} = \sup\{p^\text{min}, Q^\text{min} - r\} \) and \( T^\text{max} = \inf\{p^\text{max}, Q^\text{max} - r\} \).

Representation of the functions \( u, v \), was the input/output data for the Intel\textsuperscript{\textregistered} MKL convolution and correlation functions is described in the Data Allocation section below.

**Convolution and Correlation Data Allocation**

This section explains the relation between:

- mathematical finite functions \( u, v, w \) introduced in the section Mathematical Notation and Definitions;
- multi-dimensional input and output data vectors representing the functions \( u, v, w \);
- arrays \( u, v, w \) used to store the input and output data vectors in computer memory.

The convolution and correlation routine parameters that determine the allocation of input and output data are the following:

- Data arrays \( x, y, z \)
- Shape arrays \( xshape, yshape, zshape \)
- Strides within arrays \( xstride, ystride, zstride \)
- Parameters \( start, \text{decimation} \)

**Finite Functions and Data Vectors**

The finite functions \( u(p), v(q), \) and \( w(r) \) introduced above are represented as multi-dimensional vectors of input and output data:

\[
\begin{align*}
\text{input}(i_1, \ldots, i_{\text{dim}_u}) & \text{ for } u(p_1, \ldots, p_N) \\
\text{input}(j_1, \ldots, j_{\text{dim}_v}) & \text{ for } v(q_1, \ldots, q_N) \\
\text{output}(k_1, \ldots, k_{\text{dim}_w}) & \text{ for } w(r_1, \ldots, r_N).
\end{align*}
\]

Parameter \( \text{dims} \) represents the number of dimensions and is equal to \( N \).

The parameters \( xshape, yshape, \) and \( zshape \) define the shapes of input/output vectors:

\[
\begin{align*}
\text{input}(i_1, \ldots, i_{\text{dim}_u}) & \text{ is defined if } 1 \leq i_n \leq xshape(n) \text{ for every } n=1,\ldots,\text{dims} \\
\text{input}(j_1, \ldots, j_{\text{dim}_v}) & \text{ is defined if } 1 \leq j_n \leq yshape(n) \text{ for every } n=1,\ldots,\text{dims} \\
\text{output}(k_1, \ldots, k_{\text{dim}_w}) & \text{ is defined if } 1 \leq k_n \leq zshape(n) \text{ for every } n=1,\ldots,\text{dims}.
\end{align*}
\]

Relation between the input vectors and the functions \( u \) and \( v \) is defined by the following formulas:

\[
\begin{align*}
\text{input}(i_1, \ldots, i_{\text{dim}_u}) & = u(p_1, \ldots, p_N), \text{ where } p_n = p_n^\text{min} + (i_n-1) \text{ for every } n \\
\text{input}(j_1, \ldots, j_{\text{dim}_v}) & = v(q_1, \ldots, q_N), \text{ where } q_n = q_n^\text{min} + (j_n-1) \text{ for every } n.
\end{align*}
\]

The relation between the output vector and the function \( w(r) \) is similar (but only in the case when parameters \( start \) and \( \text{decimation} \) are not defined):

\[
\begin{align*}
\text{output}(k_1, \ldots, k_{\text{dim}_w}) & = w(r_1, \ldots, r_N), \text{ where } r_n = r_n^\text{min} + (k_n-1) \text{ for every } n.
\end{align*}
\]
If the parameter `start` is defined, it must belong to the interval \( R_{n\min} \leq start(n) \leq R_{n\max} \). If defined, the `start` parameter replaces \( R_{n\min} \) in the formula:

\[
output(k_1,\ldots,k_{\text{dims}}) = w(r_1,\ldots,r_N), \text{ where } r_n = start(n) + (k_n - 1)
\]

If the parameter `decimation` is defined, it changes the relation according to the following formula:

\[
output(k_1,\ldots,k_{\text{dims}}) = w(r_1,\ldots,r_N), \text{ where } r_n = R_{n\min} + (k_n - 1) \times \text{decimation}(n)
\]

If both parameters `start` and `decimation` are defined, the formula is as follows:

\[
output(k_1,\ldots,k_{\text{dims}}) = w(r_1,\ldots,r_N), \text{ where } r_n = \text{start}(n) + (k_n - 1) \times \text{decimation}(n)
\]

The convolution and correlation software checks the values of `zshape`, `start`, and `decimation` during task commitment. If \( r_n \) exceeds \( R_{n\max} \) for some \( k_n, n=1,\ldots,\text{dims} \), an error is raised.

**Allocation of Data Vectors**

Both parameter arrays `x` and `y` contain input data vectors in memory, while array `z` is intended for storing output data vector. To access the memory, the convolution and correlation software uses only pointers to these arrays and ignores the array shapes.

For parameters `x`, `y`, and `z`, you can provide one-dimensional arrays with the requirement that actual length of these arrays be sufficient to store the data vectors.

The allocation of the input and output data inside the arrays `x`, `y`, and `z` is described below assuming that the arrays are one-dimensional. Given multi-dimensional indices \( i, j, k \in \mathbb{Z}^N \), one-dimensional indices \( e, f, g \in \mathbb{Z} \) are defined such that:

- `input(i_1,\ldots,i_{\text{dims}})` is allocated at `x(e)`
- `input(j_1,\ldots,j_{\text{dims}})` is allocated at `y(f)`
- `output(k_1,\ldots,k_{\text{dims}})` is allocated at `z(g)`.

The indices \( e, f, \) and \( g \) are defined as follows:

\[
e = 1 + \sum x\text{stride}(n) \cdot dx(n) \quad \text{(the sum is for all } n=1,\ldots,\text{dims)}
\]

\[
f = 1 + \sum y\text{stride}(n) \cdot dy(n) \quad \text{(the sum is for all } n=1,\ldots,\text{dims)}
\]

\[
g = 1 + \sum z\text{stride}(n) \cdot dz(n) \quad \text{(the sum is for all } n=1,\ldots,\text{dims)}
\]

The distances \( dx(n), dy(n), \) and \( dz(n) \) depend on the signum of the stride:

- \( dx(n) = i_n - 1 \) if \( x\text{stride}(n) > 0 \), or \( dx(n) = i_n - x\text{shape}(n) \) if \( x\text{stride}(n) < 0 \)
- \( dy(n) = j_n - 1 \) if \( y\text{stride}(n) > 0 \), or \( dy(n) = j_n - y\text{shape}(n) \) if \( y\text{stride}(n) < 0 \)
- \( dz(n) = k_n - 1 \) if \( z\text{stride}(n) > 0 \), or \( dz(n) = k_n - z\text{shape}(n) \) if \( z\text{stride}(n) < 0 \)

The definitions of indices \( e, f, \) and \( g \) assume that indexes for arrays `x`, `y`, and `z` are started from unity:

- `x(e)` is defined for \( e=1,\ldots,\text{length}(x) \)
- `y(f)` is defined for \( f=1,\ldots,\text{length}(y) \)
- `z(g)` is defined for \( g=1,\ldots,\text{length}(z) \)

Below is a detailed explanation about how elements of the multi-dimensional output vector are stored in the array `z` for one-dimensional and two-dimensional cases.

**One-dimensional case.** If \( \text{dims}=1 \), then `zshape` is the number of the output values to be stored in the array `z`. The actual length of array `z` may be greater than `zshape` elements.
If $z_{\text{stride}}>1$, output values are stored with the stride: $\text{output}(1)$ is stored to $z(1)$, $\text{output}(2)$ is stored to $z(1+z_{\text{stride}})$, and so on. Hence, the actual length of $z$ must be at least $1+z_{\text{stride}}*(z_{\text{shape}}-1)$ elements or more.

If $z_{\text{stride}}<0$, it still defines the stride between elements of array $z$. However, the order of the used elements is the opposite. For the $k$-th output value, $\text{output}(k)$ is stored in $z(1+|z_{\text{stride}}|*(z_{\text{shape}}-k))$, where $|z_{\text{stride}}|$ is the absolute value of $z_{\text{stride}}$. The actual length of the array $z$ must be at least $1+|z_{\text{stride}}|*(z_{\text{shape}}-1)$ elements.

**Two-dimensional case.** If $\text{dims}=2$, the output data is a two-dimensional matrix. The value $z_{\text{stride}}(1)$ defines the stride inside matrix columns, that is, the stride between the $\text{output}(k_1, k_2)$ and $\text{output}(k_1+1, k_2)$ for every pair of indices $k_1$, $k_2$. On the other hand, $z_{\text{stride}}(2)$ defines the stride between columns, that is, the stride between $\text{output}(k_1,k_2)$ and $\text{output}(k_1,k_2+1)$.

If $z_{\text{stride}}(2)$ is greater than $z_{\text{shape}}(1)$, this causes sparse allocation of columns. If the value of $z_{\text{stripe}}(2)$ is smaller than $z_{\text{shape}}(1)$, this may result in the transposition of the output matrix. For example, if $z_{\text{shape}} = (2,3)$, you can define $z_{\text{stripe}} = (3,1)$ to allocate output values like transposed matrix of the shape $3\times2$.

Whether $z_{\text{stripe}}$ assumes this kind of transformations or not, you need to ensure that different elements $\text{output}(k_1, \ldots, k_{\text{dims}})$ will be stored in different locations $z(g)$.

### Summary Statistics

The Summary Statistics domain provides routines that compute basic statistical estimates for single and double precision multi-dimensional datasets.

The Summary Statistics routines calculate:

- raw and central moments up to the fourth order
- skewness and excess kurtosis (further referred to as kurtosis for brevity)
- variation coefficient
- quantiles and order statistics
- minimum and maximum
- variance-covariance/correlation matrix
- pooled/group variance-covariance matrix and mean
- partial variance-covariance/correlation matrix
- robust estimators for variance-covariance matrix and mean in presence of outliers
- raw/central partial sums up to the fourth order (for brevity referred to as raw/central sums)
- matrix of cross-products and sums of squares (for brevity referred to as cross-product matrix)
- median absolute deviation, mean absolute deviation

The library also contains functions to perform the following tasks:

- Detect outliers in datasets
- Support missing values in datasets
- Parameterize correlation matrices
- Compute quantiles for streaming data

The Mathematical Notation and Definitions section defines the supported operations in the Summary Statistics routines.

You can access the Summary Statistics routines through the Fortran 90 and C89 language interfaces. You can use the C89 interface with later versions of the C/C++.

The `mkl_vsl.h` header file is in the `$\{MKL\}/include` directory.

You can find examples that demonstrate calculation of the Summary Statistics estimates in the `$\{MKL\}/examples/vslc` example directory.
The Summary Statistics API is implemented through task objects, or tasks. A task object is a data structure, or a descriptor, holding parameters that determine a specific Summary Statistics operation. For example, such parameters may be precision, dimensions of user data, the matrix of the observations, or shapes of data arrays.

All the Summary Statistics routines process a task object as follows:

1. Create a task.
2. Modify settings of the task parameters.
3. Compute statistical estimates.
4. Destroy the task.

The Summary Statistics functions fall into the following categories:

- **Task Constructors** - routines that create a new task object descriptor and set up most common parameters (dimension, number of observations, and matrix of the observations).
- **Task Editors** - routines that can set or modify some parameter settings in the existing task descriptor.
- **Task Computation Routine** - a routine that computes specified statistical estimates.
- **Task Destructor** - a routine that deletes the task object and frees the memory.

A Summary Statistics task object contains a series of pointers to the input and output data arrays. You can read and modify the datasets and estimates at any time but you should allocate and release memory for such data.

See detailed information on the algorithms, API, and their usage in the *Intel® MKL Summary Statistics Application Notes* [SS Notes].

**Summary Statistics Naming Conventions**

The names of Summary Statistics routines, types, and constants are case-sensitive and can contain lowercase and uppercase characters (vslsSSEditQuantiles).

The names of routines have the following structure:

\[
vsl[\text{datatype}]SS<\text{base name}>\]

where

- \text{vsl} is a prefix indicating that the routine belongs to Intel® MKL Vector Statistics.
- \text{[datatype]} specifies the type of the input and/or output data and can be \text{s} (single precision real type), \text{d} (double precision real type), or \text{i} (integer type).
- \text{SS/ss} indicates that the routine is intended for calculations of the Summary Statistics estimates.
- \text{<base name>} specifies a particular functionality that the routine is designed for, for example, NewTask, Compute, DeleteTask.

**NOTE**

The Summary Statistics routine vslDeleteTask for deletion of the task is independent of the data type and its name omits the \text{[datatype]} field.

**Summary Statistics Data Types**

The Summary Statistics routines use the following data types for calculations:

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSLSSTaskPtr</td>
<td>Pointer to a Summary Statistics task</td>
</tr>
<tr>
<td>float</td>
<td>Input/output user data in single precision</td>
</tr>
</tbody>
</table>
Summary Statistics Parameters
The basic parameters in the task descriptor (addresses of dimensions, number of observations, and datasets) are assigned values when the task editors create or modify the task object. Other parameters are determined by the specific task and changed by the task editors.

Summary Statistics Task Status and Error Reporting
The task status is an integer value, which is zero if no error is detected, or a specific non-zero error code otherwise. Negative status values indicate errors, and positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The header files define symbolic names for the status codes. These names are defined as macros via `#define` statements.

The header files define the following status codes for the Summary Statistics error codes:

**Summary Statistics Status Codes**

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_STATUS_OK</td>
<td>Operation is successfully completed.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_ALLOCATION_FAILURE</td>
<td>Memory allocation has failed.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_DIMEN</td>
<td>Dimension value is invalid.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_OBSERV_N</td>
<td>Invalid number (zero or negative) of observations was obtained.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_STORAGE_NOT_SUPPORTED</td>
<td>Storage format is not supported.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_INDC_ADDR</td>
<td>Array of indices is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_WEIGHTS</td>
<td>Array of weights contains negative values.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MEAN_ADDR</td>
<td>Array of means is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_2R_MOM_ADDR</td>
<td>Array of the second order raw moments is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_3R_MOM_ADDR</td>
<td>Array of the third order raw moments is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_4R_MOM_ADDR</td>
<td>Array of the fourth order raw moments is not defined.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_2C_MOM_ADDR</td>
<td>Array of the second order central moments is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_3C_MOM_ADDR</td>
<td>Array of the third order central moments is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_4C_MOM_ADDR</td>
<td>Array of the fourth order central moments is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_KURTOSIS_ADDR</td>
<td>Array of kurtosis values is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_SKEWNESS_ADDR</td>
<td>Array of skewness values is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MIN_ADDR</td>
<td>Array of minimum values is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MAX_ADDR</td>
<td>Array of maximum values is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_VARIATION_ADDR</td>
<td>Array of variation coefficients is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_COV_ADDR</td>
<td>Covariance matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_COR_ADDR</td>
<td>Correlation matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_QUANT_ORDER_ADDR</td>
<td>Array of quantile orders is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_QUANT_ORDER</td>
<td>Quantile order value is invalid.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_QUANT_ADDR</td>
<td>Array of quantiles is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ORDER_STATS_ADDR</td>
<td>Array of order statistics is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_MOMORDER_NOT_SUPPORTED</td>
<td>Moment of requested order is not supported.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_ALL_OBSERVS_OUTLIERS</td>
<td>All observations are outliers. (At least one observation must not be an outlier.)</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ROBUST_COV_ADDR</td>
<td>Robust covariance matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ROBUST_MEAN_ADDR</td>
<td>Array of robust means is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_METHOD_NOT_SUPPORTED</td>
<td>Requested method is not supported.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_NULL_TASK_DESCRIPTOR</td>
<td>Task descriptor is null.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_OBSERV_ADDR</td>
<td>Dataset matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ACCUM_WEIGHT_ADDR</td>
<td>Pointer to the variable that holds the value of accumulated weight is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_SINGULAR_COV</td>
<td>Covariance matrix is singular.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_POOLED_COV_ADDR</td>
<td>Pooled covariance matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_POOLED_MEAN_ADDR</td>
<td>Array of pooled means is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_GROUP_COV_ADDR</td>
<td>Group covariance matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_GROUP_MEAN_ADDR</td>
<td>Array of group means is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_GROUP_INDC_ADDR</td>
<td>Array of group indices is not defined.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_GROUP_INDC</td>
<td>Group indices have improper values.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_ADDR</td>
<td>Array of parameters for the outlier detection algorithm is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_OUTLIERS_PARAMS_N_ADDR</td>
<td>Pointer to size of the parameter array for the outlier detection algorithm is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_OUTLIERS_WEIGHTS_ADDR</td>
<td>Output of the outlier detection algorithm is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_ADDR</td>
<td>Array of parameters of the robust covariance estimation algorithm is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_ROBUST_COV_PARAMS_N_ADDR</td>
<td>Pointer to the number of parameters of the algorithm for robust covariance is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STORAGE_ADDR</td>
<td>Pointer to the variable that holds the storage format is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_PARTIAL_COV_IDX_ADDR</td>
<td>Array that encodes sub-components of a random vector for the partial covariance algorithm is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_PARTIAL_COV_IDX</td>
<td>Array that encodes sub-components of a random vector for partial covariance has improper values.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_PARTIAL_COV_ADDR</td>
<td>Partial covariance matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_PARTIAL_COR_ADDR</td>
<td>Partial correlation matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_PARAMS_ADDR</td>
<td>Array of parameters for the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_PARAMS_N_ADDR</td>
<td>Pointer to number of parameters for the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_BAD_PARAMS_N</td>
<td>Size of the parameter array of the Multiple Imputation method is invalid.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_PARAMS</td>
<td>Parameters of the Multiple Imputation method are invalid.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_N_ADDR</td>
<td>Pointer to the number of initial estimates in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_INIT_ESTIMATES_ADDR</td>
<td>Array of initial estimates for the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_SIMUL_VALS_ADDR</td>
<td>Array of simulated missing values in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N_ADDR</td>
<td>Pointer to the size of the array of simulated missing values in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_ESTIMATES_N_ADDR</td>
<td>Pointer to the number of parameter estimates in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_ESTIMATES_ADDR</td>
<td>Array of parameter estimates in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_SIMUL_VALS_N</td>
<td>Invalid size of the array of simulated values in the Multiple Imputation method.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_ESTIMATES_N</td>
<td>Invalid size of an array to hold parameter estimates obtained using the Multiple Imputation method.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_OUTPUT_PARAMS</td>
<td>Array of output parameters in the Multiple Imputation method is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_PRIOR_N_ADDR</td>
<td>Pointer to the number of prior parameters is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_PRIOR_ADDR</td>
<td>Array of prior parameters is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MI_MISSING_VALS_N</td>
<td>Invalid number of missing values was obtained.</td>
</tr>
<tr>
<td>VSL_SS_SEMIDEFINITE_COR</td>
<td>Correlation matrix passed into the parameterization function is semi-definite.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_PARAMTR_COR_ADDR</td>
<td>Correlation matrix to be parameterized is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_COR</td>
<td>All eigenvalues of the correlation matrix to be parameterized are non-positive.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N_ADDR</td>
<td>Pointer to the number of parameters for the quantile computation algorithm for streaming data is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_ADDR</td>
<td>Array of parameters of the quantile computation algorithm for streaming data is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS_N</td>
<td>Invalid number of parameters of the quantile computation algorithm for streaming data has been obtained.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_PARAMS</td>
<td>Invalid parameters of the quantile computation algorithm for streaming data have been passed.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER_ADDR</td>
<td>Array of the quantile orders for streaming data is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_ORDER</td>
<td>Invalid quantile order for streaming data is defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_STREAM_QUANT_ADDR</td>
<td>Array of quantiles for streaming data is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_SUM_ADDR</td>
<td>Array of sums is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_2R_SUM_ADDR</td>
<td>Array of raw sums of 2nd order is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_3R_SUM_ADDR</td>
<td>Array of raw sums of 3rd order is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_4R_SUM_ADDR</td>
<td>Array of raw sums of 4th order is not defined.</td>
</tr>
<tr>
<td>Status Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_2C_SUM_ADDR</td>
<td>Array of central sums of 2nd order is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_3C_SUM_ADDR</td>
<td>Array of central sums of 3rd order is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_4C_SUM_ADDR</td>
<td>Array of central sums of 4th order is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_CP_SUM_ADDR</td>
<td>Cross-product matrix is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MDAD_ADDR</td>
<td>Array of median absolute deviations is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_MNAD_ADDR</td>
<td>Array of mean absolute deviations is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_BAD_SORTED_OBSERV_ADDR</td>
<td>Array for storing observation sorting results is not defined.</td>
</tr>
<tr>
<td>VSL_SS_ERROR_ERROR_INDICES_NOT_SUPPORTED</td>
<td>Array of indices is not supported.</td>
</tr>
</tbody>
</table>

Routines for robust covariance estimation, outlier detection, partial covariance estimation, multiple imputation, and parameterization of a correlation matrix can return internal error codes that are related to a specific implementation. Such error codes indicate invalid input data or other bugs in the Intel® MKL routines other than the Summary Statistics routines.

**Summary Statistics Task Constructors**

Task constructors are routines intended for creating a new task descriptor and setting up basic parameters.

**NOTE**

If the constructor fails to create a task descriptor, it returns the NULL task pointer.

**vslSSNewTask**

*Creates and initializes a new summary statistics task descriptor.*

**Syntax**

```c
status = vslSSNewTask(&task, p, n, xstorage, x, w, indices);
status = vsldSSNewTask(&task, p, n, xstorage, x, w, indices);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>const MKL_INT*</td>
<td>Dimension of the task, number of variables</td>
</tr>
</tbody>
</table>
Description

Each `vslSSNewTask` constructor routine creates a new summary statistics task descriptor with the user-specified value for a required parameter, dimension of the task. The optional parameters (matrix of observations, its storage format, number of observations, weights of observations, and indices of the random vector components) are set to their default values.

The observations of random \( p \)-dimensional vector \( \xi = (\xi_1, \ldots, \xi_p) \), which are \( n \) vectors of dimension \( p \), are passed as a one-dimensional array \( x \). The parameter \( xstorage \) defines the storage format of the observations and takes one of the possible values listed in Table "Storage format of matrix of observations and order statistics".

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_SS_MATRIX_STORAGE_ROWS</td>
<td>The observations of random vector ( \xi ) are packed by rows: ( n ) data points for the vector component ( \xi_1 ) come first, ( n ) data points for the vector component ( \xi_2 ) come second, and so forth.</td>
</tr>
<tr>
<td>VSL_SS_MATRIX_STORAGE_COLS</td>
<td>The observations of random vector ( \xi ) are packed by columns: the first ( p )-dimensional observation of the vector ( \xi ) comes first, the second ( p )-dimensional observation of the vector comes second, and so forth.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr*</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Set to VSL_STATUS_OK if the task is created successfully, otherwise a non-zero error code is returned.</td>
</tr>
</tbody>
</table>

**Storage format of matrix of observations, order statistics, and matrix of sorted observations**
A one-dimensional array \( w \) of size \( n \) contains non-negative weights assigned to the observations. You can pass a \texttt{NULL} array into the constructor. In this case, each observation is assigned the default value of the weight.

You can choose vector components for which you wish to compute statistical estimates. If an element of the vector indices of size \( p \) contains 0, the observations that correspond to this component are excluded from the calculations. If you pass the \texttt{NULL} value of the parameter into the constructor, statistical estimates for all random variables are computed.

If the constructor fails to create a task descriptor, it returns the \texttt{NULL} task pointer.

### Summary Statistics Task Editors

Task editors are intended to set up or change the task parameters listed in Table "Parameters of Summary Statistics Task to Be Initialized or Modified". As an example, to compute the sample mean for a one-dimensional dataset, initialize a variable for the mean value, and pass its address into the task as shown in the example below:

```c
#define DIM 1
#define N 1000

int main()
{
    VLSSTaskPtr task;
    double x[N];
    double mean;
    MKL_INT p, n, xstorage;
    int status;
    /* initialize variables used in the computations of sample mean */
    p = DIM;
    n = N;
    xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
    mean = 0.0;

    /* create task */
    status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, 0 );

    /* initialize task parameters */
    status = vsldSSEditTask( task, VSL_SS_ED_MEAN, &mean );

    /* compute mean using SS fast method */
    status = vsldSSCompute(task, VSL_SS_MEAN, VSL_SS_METHOD_FAST );

    /* deallocate task resources */
    status = vslSSDeleteTask( &task );

    return 0;
}
```

Use the single (\texttt{vslsssedittask}) or double (\texttt{vsldssedittask}) version of an editor, to initialize single or double precision version task parameters, respectively. Use an integer version of an editor (\texttt{vslissedittask}) to initialize parameters of the integer type.

Table "Summary Statistics Task Editors" lists the task editors for Summary Statistics. Each of them initializes and/or modifies a respective group of related parameters.
Summary Statistics Task Editors

<table>
<thead>
<tr>
<th>Editor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vslSSEditTask</code></td>
<td>Changes a pointer in the task descriptor.</td>
</tr>
<tr>
<td><code>vslSSEditMoments</code></td>
<td>Changes pointers to arrays associated with raw and central moments.</td>
</tr>
<tr>
<td><code>vslSSEditSums</code></td>
<td>Modifies the pointers to arrays that hold sum estimates.</td>
</tr>
<tr>
<td><code>vslSSEditCovCor</code></td>
<td>Changes pointers to arrays associated with covariance and/or correlation matrices.</td>
</tr>
<tr>
<td><code>vslSSEditCP</code></td>
<td>Modifies the pointers to cross-product matrix parameters.</td>
</tr>
<tr>
<td><code>vslSSEditPartialCovCor</code></td>
<td>Changes pointers to arrays associated with partial covariance and/or correlation matrices.</td>
</tr>
<tr>
<td><code>vslSSEditQuantiles</code></td>
<td>Changes pointers to arrays associated with quantile/order statistics calculations.</td>
</tr>
<tr>
<td><code>vslSSEditStreamQuantiles</code></td>
<td>Changes pointers to arrays for quantile related calculations for streaming data.</td>
</tr>
<tr>
<td><code>vslSSEditPooledCovariance</code></td>
<td>Changes pointers to arrays associated with algorithms related to a pooled covariance matrix.</td>
</tr>
<tr>
<td><code>vslSSEditRobustCovariance</code></td>
<td>Changes pointers to arrays for robust estimation of a covariance matrix and mean.</td>
</tr>
<tr>
<td><code>vslSSEditOutliersDetection</code></td>
<td>Changes pointers to arrays for detection of outliers.</td>
</tr>
<tr>
<td><code>vslSSEditMissingValues</code></td>
<td>Changes pointers to arrays associated with the method of supporting missing values in a dataset.</td>
</tr>
<tr>
<td><code>vslSSEditCorParameterization</code></td>
<td>Changes pointers to arrays associated with the algorithm for parameterization of a correlation matrix.</td>
</tr>
</tbody>
</table>

NOTE
You can use the `NULL` task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

**vslSSEditTask**

Modifies address of an input/output parameter in the task descriptor.

**Syntax**

```
status = vslSSEditTask(task, parameter, par_addr);
status = vsldSSEditTask(task, parameter, par_addr);
status = vsliSSEditTask(task, parameter, par_addr);
```

**Include Files**

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>parameter</td>
<td>const MKL_INT</td>
<td>Parameter to change</td>
</tr>
<tr>
<td>par_addr</td>
<td>const float* for vslsSSEditTask</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for vsldSSEditTask</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const MKL_INT* for vsliSSEditTask</td>
<td></td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

Description

The vslSSEditTask routine replaces the pointer to the parameter stored in the Summary Statistics task descriptor with the par_addr pointer. If you pass the NULL pointer to the editor, no changes take place in the task and a corresponding error code is returned. See Table "Parameters of Summary Statistics Task to Be Initialized or Modified" for the predefined values of the parameter.

Use the single (vslsssedittask) or double (vsldssedittask) version of the editor, to initialize single or double precision version task parameters, respectively. Use an integer version of the editor (vslissedittask) to initialize parameters of the integer type.

Parameters of Summary Statistics Task to Be Initialized or Modified

<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Type</th>
<th>Purpose</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_SS_ED_DIMEN</td>
<td>i</td>
<td>Address of a variable that holds the task dimension</td>
<td>Required. Positive integer value.</td>
</tr>
<tr>
<td>VSL_SS_ED_OBSERV_N</td>
<td>i</td>
<td>Address of a variable that holds the number of observations</td>
<td>Required. Positive integer value.</td>
</tr>
<tr>
<td>VSL_SS_ED_OBSERV</td>
<td>d, s</td>
<td>Address of the observation matrix</td>
<td>Required. Provide the matrix containing your observations.</td>
</tr>
<tr>
<td>VSL_SS_ED_OBSERV_STORAGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format for the observation matrix</td>
<td>Required. Provide a storage format supported by the library whenever you pass a matrix of observations.</td>
</tr>
<tr>
<td>VSL_SS_ED_INDC</td>
<td>i</td>
<td>Address of the array of indices</td>
<td>Optional. Provide this array if you need to process individual components of the random vector. Set entry i of the array to one to include the ith coordinate in the analysis. Set entry i of the array to zero to exclude the ith coordinate from the analysis.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_WEIGHTS</td>
<td>d, s</td>
<td>Address of the array of observation weights</td>
<td>Optional. If the observations have weights different from the default weight (one), set entries of the array to non-negative floating point values.</td>
</tr>
<tr>
<td>VSL_SS_ED_MEAN</td>
<td>d, s</td>
<td>Address of the array of means</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_2R_MOM</td>
<td>d, s</td>
<td>Address of an array of raw moments of the second order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_3R_MOM</td>
<td>d, s</td>
<td>Address of an array of raw moments of the third order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_4R_MOM</td>
<td>d, s</td>
<td>Address of an array of raw moments of the fourth order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_2C_MOM</td>
<td>d, s</td>
<td>Address of an array of central moments of the second order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.</td>
</tr>
<tr>
<td>VSL_SS_ED_3C_MOM</td>
<td>d, s</td>
<td>Address of an array of central moments of the third order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.</td>
</tr>
<tr>
<td>VSL_SS_ED_4C_MOM</td>
<td>d, s</td>
<td>Address of an array of central moments of the fourth order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
<td>--------------------------------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_KURTOSIS</td>
<td>d, s</td>
<td>Address of the array of kurtosis estimates</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, third, and fourth order.</td>
</tr>
<tr>
<td>VSL_SS_ED_SKEWNESS</td>
<td>d, s</td>
<td>Address of the array of skewness estimates</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first, second, and third order.</td>
</tr>
<tr>
<td>VSL_SS_ED_MIN</td>
<td>d, s</td>
<td>Address of the array of minimum estimates</td>
<td>Optional. Set entries of array to meaningful values, such as the values of the first observation.</td>
</tr>
<tr>
<td>VSL_SS_ED_MAX</td>
<td>d, s</td>
<td>Address of the array of maximum estimates</td>
<td>Optional. Set entries of array to meaningful values, such as the values of the first observation.</td>
</tr>
<tr>
<td>VSL_SS_ED_VARIATION</td>
<td>d, s</td>
<td>Address of the array of variation coefficients</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array. Make sure you also provide arrays for raw moments of the first and second order.</td>
</tr>
<tr>
<td>VSL_SS_ED_COV</td>
<td>d, s</td>
<td>Address of a covariance matrix</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Make sure you also provide an array for the mean.</td>
</tr>
<tr>
<td>VSL_SS_ED_COV_STORAGE</td>
<td>i</td>
<td>Address of the variable that holds the storage format for a covariance matrix</td>
<td>Required. Provide a storage format supported by the library whenever you intend to compute the covariance matrix.</td>
</tr>
</tbody>
</table>
| VSL_SS_ED_COR           | d, s | Address of a correlation matrix                   | Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a
<table>
<thead>
<tr>
<th>Parameter Value</th>
<th>Type</th>
<th>Purpose</th>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_SS_ED_COR_STORAGE</td>
<td>i</td>
<td>Address of the variable that holds the correlation storage format for a correlation matrix</td>
<td>Required. Provide a storage format supported by the library whenever you intend to compute the correlation matrix.</td>
</tr>
<tr>
<td>VSL_SS_ED_ACCUM_WEIGHT</td>
<td>d, s</td>
<td>Address of the array of size 2 that holds the accumulated weight (sum of weights) in the first position and the sum of weights squared in the second position</td>
<td>Optional. Set the entries of the matrix to meaningful values (typically zero) if you intend to do progressive processing of the dataset or need the sum of weights and sum of squared weights assigned to observations.</td>
</tr>
<tr>
<td>VSL_SS_ED_QUANT_ORDER_N</td>
<td>i</td>
<td>Address of the variable that holds the number of quantile orders</td>
<td>Required. Positive integer value. Provide the number of quantile orders whenever you compute quantiles.</td>
</tr>
<tr>
<td>VSL_SS_ED_QUANT_ORDER</td>
<td>d, s</td>
<td>Address of the array of quantile orders</td>
<td>Required. Set entries of array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.</td>
</tr>
<tr>
<td>VSL_SS_ED_QUANT_QUANTILE</td>
<td>d, s</td>
<td>Address of the array of quantiles</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_ORDER_STATS</td>
<td>d, s</td>
<td>Address of the array of order statistics</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_GROUP_INDC</td>
<td>i</td>
<td>Address of the array of group indices used in computation of a pooled covariance matrix</td>
<td>Required. Set entry $i$ to integer value $k$ if the observation belongs to group $k$. Values of $k$ take values in the range $[0, g-1]$, where $g$ is the number of groups.</td>
</tr>
<tr>
<td>VSL_SS_ED_POOLED_COV_STOR AGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format for a pooled covariance matrix</td>
<td>Required. Provide a storage format supported by the library whenever you intend to compute pooled covariance.</td>
</tr>
<tr>
<td>VSL_SS_ED_POOLED_MEAN</td>
<td>d, s</td>
<td>Address of an array of pooled means</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_POOLED_COV</td>
<td>d, s</td>
<td>Address of pooled covariance matrices</td>
<td>None.</td>
</tr>
<tr>
<td><strong>Parameter</strong></td>
<td><strong>Value</strong></td>
<td><strong>Type</strong></td>
<td><strong>Purpose</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>VSL_SS_ED_GROUP_COV_INDC</td>
<td>i</td>
<td></td>
<td>Address of an array of indices for which covariance/means should be computed</td>
</tr>
<tr>
<td>VSL_SS_ED_REQ_GROUP_INDC</td>
<td>i</td>
<td></td>
<td>Address of an array of indices for which group estimates such as covariance or means are requested</td>
</tr>
<tr>
<td>VSL_SS_ED_GROUP_MEANS</td>
<td>i</td>
<td></td>
<td>Address of an array of group means</td>
</tr>
<tr>
<td>VSL_SS_ED_GROUP_COV_STOR AGE</td>
<td>d, s</td>
<td></td>
<td>Address of a variable that holds the storage format for a group covariance matrix</td>
</tr>
<tr>
<td>VSL_SS_ED_GROUP_COV</td>
<td>d, s</td>
<td></td>
<td>Address of group covariance matrices</td>
</tr>
<tr>
<td>VSL_SS_ED_ROBUST_COV_STOR AGE</td>
<td>d, s</td>
<td></td>
<td>Address of a variable that holds the storage format for a robust covariance matrix</td>
</tr>
<tr>
<td>VSL_SS_ED_ROBUST_COV_PAR AMS_N</td>
<td>i</td>
<td></td>
<td>Address of a variable that holds the number of algorithmic parameters of the method for robust covariance estimation</td>
</tr>
<tr>
<td>VSL_SS_ED_ROBUST_COV_PAR AMS</td>
<td>d, s</td>
<td></td>
<td>Address of an array of parameters of the method for robust estimation of a covariance</td>
</tr>
<tr>
<td>VSL_SS_ED_ROBUST_MEAN</td>
<td>i</td>
<td></td>
<td>Address of an array of robust means</td>
</tr>
<tr>
<td>VSL_SS_ED_ROBUST_COV</td>
<td>d, s</td>
<td></td>
<td>Address of a robust covariance matrix</td>
</tr>
<tr>
<td>VSL_SS_ED_OUTLIERS_PARAM S_N</td>
<td>d, s</td>
<td></td>
<td>Address of a variable that holds the number of parameters of the outlier detection method</td>
</tr>
<tr>
<td>VSL_SS_ED_OUTLIERS_PARAM S</td>
<td>i</td>
<td></td>
<td>Address of an array of algorithmic parameters for the outlier detection method</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_OUTLIERS_WEIGHT</td>
<td>d, s</td>
<td>Address of an array of weights assigned to observations by the outlier detection method</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_ORDER_STATS_STORAGE</td>
<td>d, s</td>
<td>Address of a variable that holds the storage format of an order statistics matrix</td>
<td>Required. Provide a storage format supported by the library whenever you compute a matrix of order statistics.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARTIAL_COV_ID</td>
<td>i</td>
<td>Address of an array that encodes subcomponents of a random vector</td>
<td>Required. Set the entries of the array according to the description in vslSSEditPartialCovCor.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARTIAL_COV</td>
<td>d, s</td>
<td>Address of a partial covariance matrix</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARTIAL_COV_STORAGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format of a partial covariance matrix</td>
<td>Required. Provide a storage format supported by the library whenever you compute the partial covariance.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARTIAL_COR</td>
<td>d, s</td>
<td>Address of a partial correlation matrix</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARTIAL_COR_STORAGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format for a partial correlation matrix</td>
<td>Required. Provide a storage format supported by the library whenever you compute the partial correlation.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_PARAMS_N</td>
<td>i</td>
<td>Address of a variable that holds the number of algorithmic parameters for the Multiple Imputation method</td>
<td>Required. Set to the number of MI parameters, VSL_SS_MI_PARAMS_SIZE.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_PARAMS</td>
<td>d, s</td>
<td>Address of an array of algorithmic parameters for the Multiple Imputation method</td>
<td>Required. Set entries of the array according to the description in vslSSEditMissingValues.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_INIT_ESTIMES_N</td>
<td>i</td>
<td>Address of a variable that holds the number of initial estimates for the Multiple Imputation method</td>
<td>Optional. Set to $p+p^*(p+1)/2$, where $p$ is the task dimension.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_INIT_ESTIMES</td>
<td>d, s</td>
<td>Address of an array of initial estimates for the Multiple Imputation method</td>
<td>Optional. Set the values of the array according to the description in &quot;Basic Components of the Multiple Imputation Function in Summary Statistics&quot; in the Intel® MKL Summary Statistics Application Notes document [SS Notes].</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_SIMUL_VALS_ N</td>
<td>i</td>
<td>Address of a variable that holds the number of simulated values in the Multiple Imputation method</td>
<td>Optional. Positive integer indicating the number of missing points in the observation matrix.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_SIMUL_VALS d, s</td>
<td>d, s</td>
<td>Address of an array of simulated values in the Multiple Imputation method</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_ESTIMATES_N i</td>
<td>i</td>
<td>Address of a variable that holds the number of estimates obtained as a result of the Multiple Imputation method</td>
<td>Optional. Positive integer number defined according to the description in &quot;Basic Components of the Multiple Imputation Function in Summary Statistics&quot; in the Intel® MKL Summary Statistics Application Notes document [SS Notes].</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_ESTIMATES d, s</td>
<td>d, s</td>
<td>Address of an array of estimates obtained as a result of the Multiple Imputation method</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_PRIOR_N i</td>
<td>i</td>
<td>Address of a variable that holds the number of prior parameters for the Multiple Imputation method</td>
<td>Optional. If you pass a user-defined array of prior parameters, set this parameter to ((p^2+3*p+4)/2), where (p) is the task dimension.</td>
</tr>
<tr>
<td>VSL_SS_ED_MI_PRIOR d, s</td>
<td>d, s</td>
<td>Address of an array of prior parameters for the Multiple Imputation method</td>
<td>Optional. Set entries of the array of prior parameters according to the description in &quot;Basic Components of the Multiple Imputation Function in Summary Statistics&quot; in the Intel® MKL Summary Statistics Application Notes document [SS Notes].</td>
</tr>
<tr>
<td>VSL_SS_ED_PARAMTR_COR d, s</td>
<td>d, s</td>
<td>Address of a parameterized correlation matrix</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_PARAMTR_COR_STORAGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format of a parameterized correlation matrix</td>
<td>Required. Provide a storage format supported by the library whenever you compute the parameterized correlation matrix.</td>
</tr>
<tr>
<td>VSL_SS_ED_STREAM_QUANT_PARAMS_N</td>
<td>i</td>
<td>Address of a variable that holds the number of parameters of a quantile computation method for streaming data</td>
<td>Required. Set to the number of quantile computation parameters, VSL_SS_SQUANTSZW_PARAMS_N.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_STREAM_QUANT_PARAMS</td>
<td>d, s</td>
<td>Address of an array of parameters of a quantile computation method for streaming data</td>
<td>Required. Set the entries of the array according to the description in &quot;Computing Quantiles for Streaming Data&quot; in the Intel® MKL Summary Statistics Application Notes document [SS Notes].</td>
</tr>
<tr>
<td>VSL_SS_ED_STREAM_QUANT_ORDER_N</td>
<td>i</td>
<td>Address of a variable that holds the number of quantile orders for streaming data</td>
<td>Required. Positive integer value.</td>
</tr>
<tr>
<td>VSL_SS_ED_STREAM_QUANT_ORDER</td>
<td>d, s</td>
<td>Address of an array of quantile orders for streaming data</td>
<td>Required. Set entries of the array to values from the interval (0,1). Provide this parameter whenever you compute quantiles.</td>
</tr>
<tr>
<td>VSL_SS_ED_STREAM_QUANT_QUANTILES</td>
<td>d, s</td>
<td>Address of an array of quantiles for streaming data</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_SUM</td>
<td>d, s</td>
<td>Address of array of sums</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_2R_SUM</td>
<td>d, s</td>
<td>Address of array of raw sums of 2nd order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_3R_SUM</td>
<td>d, s</td>
<td>Address of array of raw sums of 3rd order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_4R_SUM</td>
<td>d, s</td>
<td>Address of array of raw sums of 4th order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_2C_SUM</td>
<td>d, s</td>
<td>Address of array of central sums of 2nd order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>Parameter Value</td>
<td>Type</td>
<td>Purpose</td>
<td>Initialization</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------</td>
<td>-------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_ED_3C_SUM</td>
<td>d, s</td>
<td>Address of array of central sums of 3rd order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_4C_SUM</td>
<td>d, s</td>
<td>Address of array of central sums of 4th order</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_CP</td>
<td>d, s</td>
<td>Address of cross-product matrix</td>
<td>Optional. Set entries of the array to meaningful values (typically zero) if you intend to compute a progressive estimate. Otherwise, do not initialize the array.</td>
</tr>
<tr>
<td>VSL_SS_ED_MDAD</td>
<td>d, s</td>
<td>Address of array of median absolute deviations</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_MNAD</td>
<td>d, s</td>
<td>Address of array of mean absolute deviations</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_SORTED_OBSERV</td>
<td>d, s</td>
<td>Address of the array that stores sorted results</td>
<td>None.</td>
</tr>
<tr>
<td>VSL_SS_ED_SORTED_OBSERV STORAGE</td>
<td>i</td>
<td>Address of a variable that holds the storage format of an output matrix</td>
<td>Required. Provide a supported storage format whenever you specify sorting of the observation matrix.</td>
</tr>
</tbody>
</table>

1. See Table: "Storage format of matrix of observations and order statistics" for storage formats.
2. See Table: "Storage formats of a variance-covariance/correlation matrix" for storage formats.

**vslSSEditMoments**

Modifies the pointers to arrays that hold moment estimates.

**Syntax**

```plaintext
status = vslsSSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
status = vsldSSEditMoments(task, mean, r2m, r3m, r4m, c2m, c3m, c4m);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
</tbody>
</table>
**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

**Description**

The `vslSSEditMoments` routine replaces pointers to the arrays that hold estimates of raw and central moments with values passed as corresponding parameters of the routine. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

**Syntax**

```c
status = vslSSEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
status = vsldSSEditSums(task, sum, r2s, r3s, r4s, c2s, c3s, c4s);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>sum</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of sums</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>r2s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of raw sums of the second order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>r3s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of raw sums of the third order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>r4s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of raw sums of the fourth order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>c2s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of central sums of the second order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>c3s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of central sums of the third order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
<tr>
<td>c4s</td>
<td>float* for vslSSEditSums</td>
<td>Pointer to the array of central sums of the fourth order</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditSums</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

**Description**

The `vslSSEditSums` routine replaces pointers to the arrays that hold estimates of raw and central sums with values passed as corresponding parameters of the routine. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

### vslSSEditCovCor

*Modifies the pointers to covariance/correlation/cross-product parameters.*

**Syntax**

```c
status = vslSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
status = vsldSSEditCovCor(task, mean, cov, cov_storage, cor, cor_storage);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
mean | float* for vslsSSEditCovCor double* for vsldSSEditCovCor | Pointer to the array of means
cov | float* for vslsSSEditCovCor double* for vsldSSEditCovCor | Pointer to a covariance matrix
cov_storage | const MKL_INT* | Pointer to the storage format of the covariance matrix
cor | float* for vslsSSEditCovCor double* for vsldSSEditCovCor | Pointer to a correlation matrix
cor_storage | const MKL_INT* | Pointer to the storage format of the correlation matrix

Output Parameters

Name | Type | Description
--- | --- | ---
status | int | Current status of the task

Description

The vslSSEditCovCor routine replaces pointers to the array of means, covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

The storage parameters, cov_storage and cor_storage, describe the storage format used for the \( p \)-by-\( p \) symmetric variance-covariance/correlation/cross-product matrix \( C \). The matrix \( C \) can be described as

\[
C = \begin{pmatrix}
c_{1,1} & c_{1,2} & \cdots & \cdots & c_{1,p} \\
c_{2,1} & c_{2,2} & \cdots & \cdots & c_{2,p} \\
: & \vdots & \ddots & \vdots & \\
: & \vdots & \ddots & \ddots & \\
c_{p,1} & c_{p,2} & \cdots & \cdots & c_{p,p}
\end{pmatrix}
\]

Table "Storage formats of a variance-covariance/correlation/cross-product matrix" shows how the matrix is stored in a one-dimensional array \( cp \) for different values of the storage parameters.

Storage formats of variance-covariance/correlation/cross-product matrices

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| VSL_SS_MATRIX_STORAGE_FULL | The array \( cp \) contains all elements of the matrix stored sequentially, row-by-row:  
\( cp[0] \) contains \( c_{1,1} \)  
\( cp[1] \) contains \( c_{1,2} \)  
\( cp[p - 1] \) contains \( c_{1,p} \)  
\( cp[p] \) contains \( c_{2,1} \)  
\( cp[p*p - 1] \) contains \( c_{p,p} \)|
The size of array \( cp \) is \( p^2 \).

**VSL_SS_MATRIX_STORAGE_L_PACKED**

The array \( cp \) contains the lower triangular part of the symmetric matrix stored sequentially, row-by-row:

- \( cp[0] \) contains \( c_{1,1} \)
- \( cp[1] \) contains \( c_{2,1} \)
- \( cp[2] \) contains \( c_{2,2} \)
  
  and so on.

The size of the array is \( p(p+1)/2 \).

**VSL_SS_MATRIX_STORAGE_U_PACKED**

The array \( cp \) contains the upper triangular part of the symmetric matrix stored sequentially, row-by-row:

- \( cp[0] \) contains \( c_{1,1} \)
- \( cp[1] \) contains \( c_{1,2} \)
- \( cp[3] \) contains \( c_{1,3} \)
  
  and so on.

The size of the array is \( p(p+1)/2 \).

### vslSSEditCP

**Modifies the pointers to cross-product matrix parameters.**

**Syntax**

```c
status = vslSSEditCP(task, mean, sum, cp, cp_storage);
status = vsldSSEditCP(task, mean, sum, cp, cp_storage);
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task</code></td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td><code>mean</code></td>
<td>float* for ( vsl)SSEditCP</td>
<td>Pointer to array of means</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditCP</td>
<td></td>
</tr>
<tr>
<td><code>sum</code></td>
<td>float* for ( vsl)SSEditCP</td>
<td>Pointer to array of sums</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditCP</td>
<td></td>
</tr>
<tr>
<td><code>cp</code></td>
<td>float* for ( vsl)SSEditCP</td>
<td>Pointer to a cross-product matrix</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditCP</td>
<td></td>
</tr>
<tr>
<td><code>cp_storage</code></td>
<td>const MKL_INT*</td>
<td>Pointer to the storage format of the cross-product matrix</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

Description

The `vslSSEditCP` routine replaces pointers to the array of means, array of sums, cross-product matrix, and its storage format with values passed as corresponding parameters of the routine. See Table: "Storage formats of a variance-covariance/correlation/cross-product matrix" for possible values of the `cp_storage` parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Storage formats of variance-covariance/correlation/cross-product matrices

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| VSL_SS_MATRIX_STORAGE_FULL | The array `cp` contains all elements of the matrix stored sequentially, row-by-row:  
  `cp[0]` contains `c1,1`  
  `cp[1]` contains `c1,2`  
  `cp[p - 1]` contains `c1,p`  
  `cp[p]` contains `c2,1`  
  `cp[p*p - 1]` contains `c_p,p`  
  The size of array `cp` is `p*p`. |
| VSL_SS_MATRIX_STORAGE_L_PACKED | The array `cp` contains the lower triangular part of the symmetric matrix stored sequentially, row-by-row:  
  `cp[0]` contains `c1,1`  
  `cp[1]` contains `c2,1`  
  `cp[2]` contains `c2,2`  
  and so on.  
  The size of the array is `p*(p+ 1)/2`. |
| VSL_SS_MATRIX_STORAGE_U_PACKED | The array `cp` contains the upper triangular part of the symmetric matrix stored sequentially, row-by-row:  
  `cp[0]` contains `c1,1`  
  `cp[1]` contains `c1,2`  
  `cp[3]` contains `c1,3`  
  and so on.  
  The size of the array is `p*(p+ 1)/2`. |

`vslSSEditPartialCovCor`

Modifies the pointers to partial covariance/correlation parameters.

Syntax

```c
status = vslSSEditPartialCovCor(task, p_idx_array, cov, cov_storage, cor, cor_storage, p_cov, p_cov_storage, p_cor, p_cor_storage);
```
\texttt{status} = \texttt{vslSSEditPartialCovCor}(\texttt{task}, \texttt{p_idx_array}, \texttt{cov}, \texttt{cov_storage}, \texttt{cor}, \texttt{cor_storage},
\texttt{p_cov}, \texttt{p_cov_storage}, \texttt{p_cor}, \texttt{p_cor_storage});

\textbf{Include Files}
- \texttt{mkl.h}

\textbf{Input Parameters}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{task}</td>
<td>\texttt{VSLSSTaskPtr}</td>
<td>Descriptor of the task</td>
</tr>
</tbody>
</table>
| \texttt{p_idx_array} | \texttt{const MKL_INT*}  | Pointer to the array that encodes indices of subcomponents \(Z\) and \(Y\) of the random vector as described in section \textit{Mathematical Notation and Definitions}. \(p_{idx\ array}[i]\) equals to 
-1 if the \(i\)-th component of the random vector belongs to \(Z\)
1, if the \(i\)-th component of the random vector belongs to \(Y\). |
| \texttt{cov}    | \texttt{const float* for vslSSEditPartialCovCor} \texttt{const double* for vsldSSEditPartialCovCor} | Pointer to a covariance matrix |
| \texttt{cov_storage} | \texttt{const MKL_INT*}  | Pointer to the storage format of the covariance matrix |
| \texttt{cor}    | \texttt{const float* for vslSSEditPartialCovCor} \texttt{const double* for vsldSSEditPartialCovCor} | Pointer to a correlation matrix |
| \texttt{cor_storage} | \texttt{const MKL_INT*}  | Pointer to the storage format of the correlation matrix |
| \texttt{p_cov}  | \texttt{float* for vslSSEditPartialCovCor} \texttt{double* for vsldSSEditPartialCovCor} | Pointer to a partial covariance matrix |
| \texttt{p_cov_storage} | \texttt{const MKL_INT*}  | Pointer to the storage format of the partial covariance matrix |
| \texttt{p_cor}  | \texttt{float* for vslSSEditPartialCovCor} \texttt{double* for vsldSSEditPartialCovCor} | Pointer to a partial correlation matrix |
| \texttt{p_cor_storage} | \texttt{const MKL_INT*}  | Pointer to the storage format of the partial correlation matrix |
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

Description

The `vslSSEditPartialCovCor` routine replaces pointers to covariance/correlation arrays, partial covariance/correlation arrays, and their storage format with values passed as corresponding parameters of the routine. See Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the `cov_storage`, `cor_storage`, `p_cov_storage`, and `p_cor_storage` parameters. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

`vslSSEditQuantiles`

*Modifies the pointers to parameters related to quantile computations.*

Syntax

```c
status = vslSSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats, order_stats_storage);
status = vsldSSEditQuantiles(task, quant_order_n, quant_order, quants, order_stats, order_stats_storage);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>quant_order_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of quantile orders</td>
</tr>
<tr>
<td>quant_order</td>
<td>const float* for vslSSEditQuantiles</td>
<td>Pointer to the array of quantile orders</td>
</tr>
<tr>
<td></td>
<td>const double* for vsldSSEditQuantiles</td>
<td></td>
</tr>
<tr>
<td>quants</td>
<td>float* for vslSSEditQuantiles</td>
<td>Pointer to the array of quantiles</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditQuantiles</td>
<td></td>
</tr>
<tr>
<td>order_stats</td>
<td>float* for vslSSEditQuantiles</td>
<td>Pointer to the array of order statistics</td>
</tr>
<tr>
<td></td>
<td>double* for vsldSSEditQuantiles</td>
<td></td>
</tr>
<tr>
<td>order_stats_storage</td>
<td>const MKL_INT*</td>
<td>Pointer to the storage format of the order statistics array</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

**Description**

The `vslSSEditQuantiles` routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the array that holds order statistics, and the storage format for the order statistics with values passed into the routine. See Table "Storage format of matrix of observations and order statistics" for possible values of the `order_statistics_storage` parameter. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

**vslSSEditStreamQuantiles**

*Modifies the pointers to parameters related to quantile computations for streaming data.*

**Syntax**

```c
status = vslsSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams, params);
status = vsldSSEditStreamQuantiles(task, quant_order_n, quant_order, quants, nparams, params);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>quant_order_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of quantile orders</td>
</tr>
<tr>
<td>quant_order</td>
<td>const float* for vslsSSEditStreamQuantiles</td>
<td>Pointer to the array of quantile orders</td>
</tr>
<tr>
<td>quants</td>
<td>float* for vslsSSEditStreamQuantiles</td>
<td>Pointer to the array of quantiles</td>
</tr>
<tr>
<td>nparams</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of the algorithm parameters</td>
</tr>
<tr>
<td>params</td>
<td>const float* for vslsSSEditStreamQuantiles</td>
<td>Pointer to the array of the algorithm parameters</td>
</tr>
</tbody>
</table>
The vsdSSEditStreamQuantiles routine replaces pointers to the number of quantile orders, the array of quantile orders, the array of quantiles, the number of the algorithm parameters, and the array of the algorithm parameters with values passed into the routine. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

vsdSSEditPooledCovariance

Modifies pooled/group covariance matrix array pointers.

Syntax

\[
\text{status} = \text{vsdSSEditPooledCovariance}(\text{task}, \text{grp\_indices}, \text{pld\_mean}, \text{pld\_cov}, \\
\text{req\_grp\_indices}, \text{grp\_means}, \text{grp\_cov});
\]

\[
\text{status} = \text{vsldSSEditPooledCovariance}(\text{task}, \text{grp\_indices}, \text{pld\_mean}, \text{pld\_cov}, \\
\text{req\_grp\_indices}, \text{grp\_means}, \text{grp\_cov});
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>grp_indices</td>
<td>const MKL_INT*</td>
<td>Pointer to an array of size ( n ). The ( i )-th element of the array contains the number of the group the observation belongs to.</td>
</tr>
<tr>
<td>pld_mean</td>
<td>float* for vsdSSEditPooledCovariance</td>
<td>Pointer to the array of pooled means</td>
</tr>
<tr>
<td>pld_cov</td>
<td>float* for vsdSSEditPooledCovariance</td>
<td>Pointer to the array that holds a pooled covariance matrix</td>
</tr>
<tr>
<td>req_grp_indices</td>
<td>const MKL_INT*</td>
<td>Pointer to the array that contains indices of groups for which estimates to return (such as covariance and mean)</td>
</tr>
</tbody>
</table>
### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

### Description

The `vslSSEditPooledCovariance` routine replaces pointers to the array of group indices, the array of pooled means, the array for a pooled covariance matrix, and pointers to the array of indices of group matrices, the array of group means, and the array for group covariance matrices with values passed in the editors. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged. Use the `vslSSEditTask` routine to replace the storage format for pooled and group covariance matrices.

### vslSSEditRobustCovariance

**Modifies pointers to arrays related to a robust covariance matrix.**

**Syntax**

```c
status = vslSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
status = vsldSSEditRobustCovariance(task, rcov_storage, nparams, params, rmean, rcov);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>rcov_storage</td>
<td>const MKL_INT*</td>
<td>Pointer to the storage format of a robust covariance matrix</td>
</tr>
<tr>
<td>nparams</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of method parameters</td>
</tr>
<tr>
<td>params</td>
<td>const float* for</td>
<td>Pointer to the array of method parameters</td>
</tr>
<tr>
<td></td>
<td>vslSSEditRobustCovariance</td>
<td></td>
</tr>
</tbody>
</table>
Name | Type | Description
--- | --- | ---
| const double* | for vsldSSEditRobustCovariance | 

\( rmean \)

| float* | for vslsSSEditRobustCovariance | Pointer to the array of robust means
| double* | for vsldSSEditRobustCovariance |

\( rcov \)

| float* | for vslsSSEditRobustCovariance | Pointer to a robust covariance matrix
| double* | for vsldSSEditRobustCovariance |

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

### Description

The `vslSSEditRobustCovariance` routine uses values passed as parameters of the routine to replace:

- pointers to covariance matrix storage
- pointers to the number of method parameters and to the array of the method parameters of size `nparams`
- pointers to the arrays that hold robust means and covariance

See Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the `rcov_storage` parameter. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® MKL provides a Translated Biweight S-estimator (TBS) for robust estimation of a variance-covariance matrix and mean [Rocke96]. Use one iteration of the Maronna algorithm with the reweighting step [Maronna02] to compute the initial point of the algorithm. Pack the parameters of the TBS algorithm into the `params` array and pass them into the editor. Table "Structure of the Array of TBS Parameters" describes the `params` structure.

### Structure of the Array of TBS Parameters

<table>
<thead>
<tr>
<th>Array Position</th>
<th>Algorithm Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \varepsilon )</td>
<td>Breakdown point, the number of outliers the algorithm can hold. By default, the value is ( (n-p)/(2n) ).</td>
</tr>
<tr>
<td>1</td>
<td>( \alpha )</td>
<td>Asymptotic rejection probability, see details in [Rocke96]. By default, the value is 0.001.</td>
</tr>
<tr>
<td>2</td>
<td>( \delta )</td>
<td>Stopping criterion: the algorithm is terminated if weights are changed less than ( \delta ). By default, the value is 0.001.</td>
</tr>
<tr>
<td>3</td>
<td>max_iter</td>
<td>Maximum number of iterations. The algorithm terminates after <code>max_iter</code> iterations. By default, the value is 10.</td>
</tr>
</tbody>
</table>
If you set this parameter to zero, the function returns a robust estimate of the variance-covariance matrix computed using the Maronna method [Maronna02] only.

The robust estimator of variance-covariance implementation in Intel® MKL requires that the number of observations $n$ be greater than twice the number of variables: $n > 2p$.

See additional details of the algorithm usage model in the Intel® MKL Summary Statistics Application Notes document [SS Notes].

### vslSSEditOutliersDetection

**Modifies array pointers related to multivariate outliers detection.**

**Syntax**

```c
status = vslsSSEditOutliersDetection(task, nparams, params, w);
status = vsldSSEditOutliersDetection(task, nparams, params, w);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>nparams</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of method parameters</td>
</tr>
<tr>
<td>params</td>
<td>const float* for</td>
<td>Pointer to the array of method parameters</td>
</tr>
<tr>
<td></td>
<td>vslsSSEditOutliersDetection</td>
<td></td>
</tr>
<tr>
<td></td>
<td>const double* for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vsldSSEditOutliersDetection</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>float* for</td>
<td>Pointer to an array of size $n$. The array holds</td>
</tr>
<tr>
<td></td>
<td>vslsSSEditOutliersDetection</td>
<td>the weights of observations to be marked as</td>
</tr>
<tr>
<td></td>
<td></td>
<td>outliers.</td>
</tr>
<tr>
<td></td>
<td>double* for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>vsldSSEditOutliersDetection</td>
<td></td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

**Description**

The `vslSSEditOutliersDetection` routine uses the parameters passed to replace
the pointers to the number of method parameters and to the array of the method parameters of size \( n \text{params} \)
• the pointer to the array that holds the calculated weights of the observations

If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Intel® MKL provides the BACON algorithm ([Billor00]) for the detection of multivariate outliers. Pack the parameters of the BACON algorithm into the \( \text{params} \) array and pass them into the editor. Table "Structure of the Array of BACON Parameters" describes the \( \text{params} \) structure.

Structure of the Array of BACON Parameters

<table>
<thead>
<tr>
<th>Array Position</th>
<th>Algorithm Parameter</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0              | Method to start the algorithm | The parameter takes one of the following possible values: 

\[
\text{VSL_SS_METHOD_BACON_MEDIAN_INIT}, \text{if the algorithm is started using the median estimate. This is the default value of the parameter.}
\]

\[
\text{VSL_SS_METHOD_BACON_MAHALANOBIS_INIT}, \text{if the algorithm is started using the Mahalanobis distances.}
\]

| 1              | \( \alpha \)         | One-tailed probability that defines the \((1 - \alpha)\) quantile of \( \chi^2 \) distribution with \( p \) degrees of freedom. The recommended value is \( \alpha/n \), where \( n \) is the number of observations. By default, the value is 0.05. |
| 2              | \( \delta \)          | Stopping criterion; the algorithm is terminated if the size of the basic subset is changed less than \( \delta \). By default, the value is 0.005. |

Output of the algorithm is the vector of weights, \( \text{BaconWeights} \), such that \( \text{BaconWeights}(i) = 0 \) if \( i \)-th observation is detected as an outlier. Otherwise \( \text{BaconWeights}(i) = w(i) \), where \( w \) is the vector of input weights. If you do not provide the vector of input weights, \( \text{BaconWeights}(i) \) is set to 1 if the \( i \)-th observation is not detected as an outlier.

See additional details about usage model of the algorithm in the Intel® MKL Summary Statistics Application Notes document [SS Notes].

\text{vsISSEditMissingValues}

Modified pointers to arrays associated with the method of supporting missing values in a dataset.

Syntax

\[
\begin{align*}
\text{status} = \text{vsISSEditMissingValues}(\text{task, nparams, params, init_estimates_n, init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n, estimates}); \\
\text{status} = \text{vsldSSEditMissingValues}(\text{task, nparams, params, init_estimates_n, init_estimates, prior_n, prior, simul_missing_vals_n, simul_missing_vals, estimates_n, estimates});
\end{align*}
\]

Include Files

• mkl.h
### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>nparams</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of method parameters</td>
</tr>
<tr>
<td>params</td>
<td>const float* for vslsSSEditMissingValues for vslsSSEditMissingValues</td>
<td>Pointer to the array of method parameters</td>
</tr>
<tr>
<td>init_estimates_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of initial estimates for mean and a variance-covariance matrix</td>
</tr>
<tr>
<td>init_estimates</td>
<td>const float* for vslsSSEditMissingValues for vslsSSEditMissingValues</td>
<td>Pointer to the array that holds initial estimates for mean and a variance-covariance matrix</td>
</tr>
<tr>
<td>prior_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of prior parameters</td>
</tr>
<tr>
<td>prior</td>
<td>const float* for vslsSSEditMissingValues for vslsSSEditMissingValues</td>
<td>Pointer to the array of prior parameters</td>
</tr>
<tr>
<td>simul_missing_vals_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the size of the array that holds output of the Multiple Imputation method</td>
</tr>
<tr>
<td>simul_missing_vals</td>
<td>float* for vslsSSEditMissingValues for vslsSSEditMissingValues</td>
<td>Pointer to the array of size $k \times m$, where $k$ is the total number of missing values, and $m$ is number of copies of missing values. The array holds $m$ sets of simulated missing values for the matrix of observations.</td>
</tr>
<tr>
<td>estimates_n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of estimates to be returned by the routine</td>
</tr>
<tr>
<td>estimates</td>
<td>float* for vslsSSEditMissingValues for vslsSSEditMissingValues</td>
<td>Pointer to the array that holds estimates of the mean and a variance-covariance matrix.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

Description

The `vslSSEditMissingValues` routine uses values passed as parameters of the routine to replace pointers to the number and the array of the method parameters, pointers to the number and the array of initial mean/variance-covariance estimates, the pointer to the number and the array of prior parameters, pointers to the number and the array of simulated missing values, and pointers to the number and the array of the intermediate mean/covariance estimates. If you pass a value of `NULL` for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

Before you call the Summary Statistics routines to process missing values, preprocess the dataset and denote missing observations with one of the following predefined constants:

- `VSL_SS_SNAN`, if the dataset is stored in single precision floating-point arithmetic
- `VSL_SS_DNAN`, if the dataset is stored in double precision floating-point arithmetic

Intel® MKL provides the `VSL_SS_METHOD_MI` method to support missing values in the dataset based on the Multiple Imputation (MI) approach described in [Schafer97]. The following components support Multiple Imputation:

- Expectation Maximization (EM) algorithm to compute the start point for the Data Augmentation (DA) procedure
- DA function

NOTE

The DA component of the MI procedure is simulation-based and uses the `VSL_BRNG_MCG59` basic random number generator with predefined `seed = 2^{50}` and the Gaussian distribution generator (ICDF method) available in Intel® MKL [Gaussian].

Pack the parameters of the MI algorithm into the `params` array. Table "Structure of the Array of MI Parameters" describes the `params` structure.

Structure of the Array of MI Parameters

<table>
<thead>
<tr>
<th>Array Position</th>
<th>Algorithm Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>em_iter_num</td>
<td>Maximal number of iterations for the EM algorithm. By default, this value is 50.</td>
</tr>
<tr>
<td>1</td>
<td>da_iter_num</td>
<td>Maximal number of iterations for the DA algorithm. By default, this value is 30.</td>
</tr>
<tr>
<td>2</td>
<td>ε</td>
<td>Stopping criterion for the EM algorithm. The algorithm terminates if the maximal module of the element-wise difference between the previous and current parameter values is less than ε. By default, this value is 0.001.</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>Number of sets to impute</td>
</tr>
<tr>
<td>4</td>
<td>missing_vals_num</td>
<td>Total number of missing values in the datasets</td>
</tr>
</tbody>
</table>
You can also pass initial estimates into the EM algorithm by packing both the vector of means and the variance-covariance matrix as a one-dimensional array `init_estimates`. The size of the array should be at least \( p + p(p + 1)/2 \). For \( i=0, \ldots, p-1 \), the `init_estimates[i]` array contains the initial estimate of means. The remaining positions of the array are occupied by the upper triangular part of the variance-covariance matrix.

If you provide no initial estimates for the EM algorithm, the editor uses the default values, that is, the vector of zero means and the unitary matrix as a variance-covariance matrix. You can also pass prior parameters for \( \mu \) and \( \Sigma \) into the library: \( \mu_0, \tau, m, \) and \( \Lambda^{-1} \). Pack these parameters as a one-dimensional array `prior` with a size of at least \( (p^2 + 3p + 4)/2 \).

The storage format is as follows:

- `prior[0], \ldots, prior[p-1]` contain the elements of the vector \( \mu_0 \).
- `prior[p]` contains the parameter \( \tau \).
- `prior[p+1]` contains the parameter \( m \).
- The remaining positions are occupied by the upper-triangular part of the inverted matrix \( \Lambda^{-1} \).

If you provide no prior parameters, the editor uses their default values:

- The array of \( p \) zeros is used as \( \mu_0 \).
- \( \tau \) is set to 0.
- \( m \) is set to \( p \).
- The zero matrix is used as an initial approximate of \( \Lambda^{-1} \).

The `EditMissingValues` editor returns \( m \) sets of imputed values and/or a sequence of parameter estimates drawn during the DA procedure.

The editor returns the imputed values as the `simul_missing_vals` array. The size of the array should be sufficient to hold \( m \) sets each of the `missing_vals_num` size, that is, at least \( m \times \text{missing_vals_num} \) in total. The editor packs the imputed values one by one in the order of their appearance in the matrix of observations.

For example, consider a task of dimension 4. The total number of observations \( n \) is 10. The second observation vector misses variables 1 and 2, and the seventh observation vector lacks variable 1. The number of sets to impute is \( m=2 \). Then, `simul_missing_vals[0]` and `simul_missing_vals[1]` contains the first and the second points for the second observation vector, and `simul_missing_vals[2]` holds the first point for the seventh observation. Positions 3, 4, and 5 are formed similarly.

To estimate convergence of the DA algorithm and choose a proper value of the number of DA iterations, request the sequence of parameter estimates that are produced during the DA procedure. The editor returns the sequence of parameters as a single array. The size of the array is

\[
m \times da_iter_num \times (p + (p^2 + p)/2)
\]

where

- \( m \) is the number of sets of values to impute.
- `da_iter_num` is the number of DA iterations.
- The value \( p + (p^2 + p)/2 \) determines the size of the memory to hold one set of the parameter estimates.

In each set of the parameters, the vector of means occupies the first \( p \) positions and the remaining \( (p^2 + p)/2 \) positions are intended for the upper triangular part of the variance-covariance matrix.

Upon successful generation of \( m \) sets of imputed values, you can place them in cells of the data matrix with missing values and use the Summary Statistics routines to analyze and get estimates for each of the \( m \) complete datasets.
NOTE
Intel® MKL implementation of the MI algorithm rewrites cells of the dataset that contain the VSL_SS_SNAN/VSL_SS_DNAN values. If you want to use the Summary Statistics routines to process the data with missing values again, mask the positions of the empty cells.

See additional details of the algorithm usage model in the Intel® MKL Summary Statistics Application Notes document [SS Notes].

vslSSEditCorParameterization
Modifies pointers to arrays related to the algorithm of correlation matrix parameterization.

Syntax

status = vslSSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);
status = vsldSSEditCorParameterization(task, cor, cor_storage, pcor, pcor_storage);

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>cor</td>
<td>const float* for vslsSSEditCorParameterization, const double* for vsldSSEditCorParameterization</td>
<td>Pointer to the correlation matrix</td>
</tr>
<tr>
<td>cor_storage</td>
<td>const MKL_INT*</td>
<td>Pointer to the storage format of the correlation matrix</td>
</tr>
<tr>
<td>pcor</td>
<td>float* for vslsSSEditCorParameterization, double* for vsldSSEditCorParameterization</td>
<td>Pointer to the parameterized correlation matrix</td>
</tr>
<tr>
<td>por_storage</td>
<td>const MKL_INT*</td>
<td>Pointer to the storage format of the parameterized correlation matrix</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

Description

The vslSSEditCorParameterization routine uses values passed as parameters of the routine to replace pointers to the correlation matrix, pointers to the correlation matrix storage format, a pointer to the parameterized correlation matrix, and a pointer to the parameterized correlation matrix storage format. See
Table "Storage formats of a variance-covariance/correlation matrix" for possible values of the cor_storage and pcor_storage parameters. If you pass a value of NULL for a specific input parameter, the value of that parameter in the task descriptor is unchanged.

**Summary Statistics Task Computation Routines**

Task computation routines calculate statistical estimates on the data provided and parameters held in the task descriptor. After you create the task and initialize its parameters, you can call the computation routines as many times as necessary. Table "Summary Statistics Estimates Obtained with vslSSCompute Routine" lists the statistical estimates that you can obtain using the vslSSCompute routine.

**NOTE**
The Summary Statistics computation routines do not signal floating-point errors, such as overflow or gradual underflow, or operations with NaNs (except for the missing values in the observations).

### Summary Statistics Estimates Obtained with vslSSCompute Routine

<table>
<thead>
<tr>
<th>Estimate</th>
<th>Support of Observations Available in Blocks</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_SS_MEAN</td>
<td>Yes</td>
<td>Computes the array of means.</td>
</tr>
<tr>
<td>VSL_SS_SUM</td>
<td>Yes</td>
<td>Computes the array of sums.</td>
</tr>
<tr>
<td>VSL_SS_2R_MOM</td>
<td>Yes</td>
<td>Computes the array of the 2\textsuperscript{nd} order raw moments.</td>
</tr>
<tr>
<td>VSL_SS_2R_SUM</td>
<td>Yes</td>
<td>Computes the array of raw sums of the 2\textsuperscript{nd} order.</td>
</tr>
<tr>
<td>VSL_SS_3R_MOM</td>
<td>Yes</td>
<td>Computes the array of the 3\textsuperscript{rd} order raw moments.</td>
</tr>
<tr>
<td>VSL_SS_3R_SUM</td>
<td>Yes</td>
<td>Computes the array of raw sums of the 3\textsuperscript{rd} order.</td>
</tr>
<tr>
<td>VSL_SS_4R_MOM</td>
<td>Yes</td>
<td>Computes the array of the 4\textsuperscript{th} order raw moments.</td>
</tr>
<tr>
<td>VSL_SS_4R_SUM</td>
<td>Yes</td>
<td>Computes the array of raw sums of the 4\textsuperscript{th} order.</td>
</tr>
<tr>
<td>VSL_SS_2C_MOM</td>
<td>Yes</td>
<td>Computes the array of the 2\textsuperscript{nd} order central moments.</td>
</tr>
<tr>
<td>VSL_SS_2C_SUM</td>
<td>Yes</td>
<td>Computes the array of central sums of the 2\textsuperscript{nd} order.</td>
</tr>
<tr>
<td>VSL_SS_3C_MOM</td>
<td>Yes</td>
<td>Computes the array of the 3\textsuperscript{rd} order central moments.</td>
</tr>
<tr>
<td>VSL_SS_3C_SUM</td>
<td>Yes</td>
<td>Computes the array of central sums of the 3\textsuperscript{rd} order.</td>
</tr>
<tr>
<td>VSL_SS_4C_MOM</td>
<td>Yes</td>
<td>Computes the array of the 4\textsuperscript{th} order central moments.</td>
</tr>
<tr>
<td>VSL_SS_4C_SUM</td>
<td>Yes</td>
<td>Computes the array of central sums of the 4\textsuperscript{th} order.</td>
</tr>
<tr>
<td>Estimate</td>
<td>Support of Observations Available in Blocks</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>VSL_SS_KURTOSIS</td>
<td>Yes</td>
<td>Computes the array of kurtosis values.</td>
</tr>
<tr>
<td>VSL_SS_SKEWNESS</td>
<td>Yes</td>
<td>Computes the array of skewness values.</td>
</tr>
<tr>
<td>VSL_SS_MIN</td>
<td>Yes</td>
<td>Computes the array of minimum values.</td>
</tr>
<tr>
<td>VSL_SS_MAX</td>
<td>Yes</td>
<td>Computes the array of maximum values.</td>
</tr>
<tr>
<td>VSL_SS_VARIATION</td>
<td>Yes</td>
<td>Computes the array of variation coefficients.</td>
</tr>
<tr>
<td>VSL_SS_COV</td>
<td>Yes</td>
<td>Computes a covariance matrix.</td>
</tr>
<tr>
<td>VSL_SS_COR</td>
<td>Yes</td>
<td>Computes a correlation matrix. The main diagonal of the correlation matrix holds variances of the random vector components.</td>
</tr>
<tr>
<td>VSL_SS_CP</td>
<td>Yes</td>
<td>Computes a cross-product matrix.</td>
</tr>
<tr>
<td>VSL_SS_POOLED_COV</td>
<td>No</td>
<td>Computes a pooled covariance matrix.</td>
</tr>
<tr>
<td>VSL_SS_POOLED_MEAN</td>
<td>No</td>
<td>Computes an array of pooled means.</td>
</tr>
<tr>
<td>VSL_SS_GROUP_COV</td>
<td>No</td>
<td>Computes group covariance matrices.</td>
</tr>
<tr>
<td>VSL_SS_GROUP_MEAN</td>
<td>No</td>
<td>Computes group means.</td>
</tr>
<tr>
<td>VSL_SS_QUANTS</td>
<td>No</td>
<td>Computes quantiles.</td>
</tr>
<tr>
<td>VSL_SS_ORDER_STATS</td>
<td>No</td>
<td>Computes order statistics.</td>
</tr>
<tr>
<td>VSL_SS_ROBUST_COV</td>
<td>No</td>
<td>Computes a robust covariance matrix.</td>
</tr>
<tr>
<td>VSL_SS_OUTLIERS</td>
<td>No</td>
<td>Detects outliers in the dataset.</td>
</tr>
<tr>
<td>VSL_SS_PARTIAL_COV</td>
<td>No</td>
<td>Computes a partial covariance matrix.</td>
</tr>
<tr>
<td>VSL_SS_PARTIAL_COR</td>
<td>No</td>
<td>Computes a partial correlation matrix.</td>
</tr>
<tr>
<td>VSL_SS_MISSING_VALS</td>
<td>No</td>
<td>Supports missing values in datasets.</td>
</tr>
<tr>
<td>VSL_SS_PARAMTR_COR</td>
<td>No</td>
<td>Computes a parameterized correlation matrix.</td>
</tr>
<tr>
<td>VSL_SS_STREAM_QUANTS</td>
<td>Yes</td>
<td>Computes quantiles for streaming data.</td>
</tr>
<tr>
<td>VSL_SS_MDAD</td>
<td>No</td>
<td>Computes median absolute deviation.</td>
</tr>
<tr>
<td>VSL_SS_MNAD</td>
<td>No</td>
<td>Computes mean absolute deviation.</td>
</tr>
<tr>
<td>VSL_SS_SORTED_OBSERV</td>
<td>No</td>
<td>Sorts the dataset by the components of the random vector ( \xi ).</td>
</tr>
</tbody>
</table>

Table "Summary Statistics Computation Method" lists estimate calculation methods supported by Intel® MKL. See the Intel® MKL Summary Statistics Application Notes document [SS Notes] for a detailed description of the methods.

**Summary Statistics Computation Method**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VSL_SS_METHOD_FAST</td>
<td>Fast method for calculation of the estimates:</td>
</tr>
</tbody>
</table>
Method | Description
--- | ---
| • raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix  
• min/max/quantile/order statistics  
• partial variance-covariance  
• median/mean absolute deviation  
VSL_SS_METHOD_FAST_USER_MEAN | Fast method for calculation of the estimates given user-defined mean:  
• central moments/sums of 2-4 order, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix, mean absolute deviation  
VSL_SS_METHOD_1PASS | One-pass method for calculation of estimates:  
• raw/central moments/sums, skewness, kurtosis, variation, variance-covariance/correlation/cross-product matrix  
• pooled/group covariance matrix  
VSL_SS_METHOD_TBS | TBS method for robust estimation of covariance and mean  
VSL_SS_METHOD_BACON | BACON method for detection of multivariate outliers  
VSL_SS_METHOD_MI | Multiple imputation method for support of missing values  
VSL_SS_METHOD_SD | Spectral decomposition method for parameterization of a correlation matrix  
VSL_SS_METHOD_SQUANTS_ZW | Zhang-Wang (ZW) method for quantile estimation for streaming data  
VSL_SS_METHOD_SQUANTS_ZW_FAST | Fast ZW method for quantile estimation for streaming data  
VSL_SS_METHOD_RADIX | Radix method for dataset sorting

You can calculate all requested estimates in one call of the routine. For example, to compute a kurtosis and covariance matrix using a fast method, pass a combination of the pre-defined parameters into the Compute routine as shown in the example below:

```c
...  
method = VSL_SS_METHOD_FAST;  
task_params = VSL_SS_KURTOSIS|VSL_SS_COV;  
...  
status = vsldSSCompute(task, task_params, method);
```

To compute statistical estimates for the next block of observations, you can do one of the following:

- copy the observations to memory, starting with the address available to the task
- use one of the appropriate Editors to modify the pointer to the new dataset in the task.

The library does not detect your changes of the dataset and computed statistical estimates. To obtain statistical estimates for a new matrix, change the observations and initialize relevant arrays. You can follow this procedure to compute statistical estimates for observations that come in portions. See Table "Summary Statistics Estimates Obtained with vslSSCompute Routine" for information on such observations supported by the Intel® MKL Summary Statistics estimators.
To modify parameters of the task using the Task Editors, set the address of the targeted matrix of the observations or change the respective vector component indices. After you complete editing the task parameters, you can compute statistical estimates in the modified environment.

If the task completes successfully, the computation routine returns the zero status code. If an error is detected, the computation routine returns an error code. In particular, an error status code is returned in the following cases:

- the task pointer is NULL
- memory allocation has failed
- the calculation has failed for some other reason

**NOTE**
You can use the NULL task pointer in calls to editor routines. In this case, the routine is terminated and no system crash occurs.

### vslSSCompute
*Computes Summary Statistics estimates.*

**Syntax**

```c
status = vslsSSCompute(task, estimates, method);
status = vsldSSCompute(task, estimates, method);
```

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>VSLSSTaskPtr</td>
<td>Descriptor of the task</td>
</tr>
<tr>
<td>estimates</td>
<td>const MKL_INT64</td>
<td>List of statistical estimates to compute</td>
</tr>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Method to be used in calculations</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Current status of the task</td>
</tr>
</tbody>
</table>

**Description**

The `vslSSCompute` routine calculates statistical estimates passed as the `estimates` parameter using the algorithms passed as the `method` parameter of the routine. The computations are done in the context of the task descriptor that contains pointers to all required and optional, if necessary, properly initialized arrays. In one call of the function, you can compute several estimates using proper methods for their calculation. See Table "Summary Statistics Estimates Obtained with Compute Routine" for the list of the estimates that you can calculate with the `vslSSCompute` routine. See Table "Summary Statistics Computation Methods" for the list of possible values of the `method` parameter.

To initialize single or double precision version task parameters, use the single (`vslsscompute`) or double (`vsldsscompute`) version of the editor, respectively. To initialize parameters of the integer type, use an integer version of the editor (`vslisscompute`).

8 Intel® Math Kernel Library Developer Reference 2182
NOTE
Requesting a combination of the \texttt{VSL_SS_MISSING_VALS} value and any other estimate parameter in the \texttt{Compute} function results in processing only the missing values.

Application Notes

Be aware that when computing a correlation matrix, the \texttt{vslSSCompute} routine allocates an additional array for each thread which is running the task. If you are running on a large number of threads \texttt{vslSSCompute} might consume large amounts of memory.

When calculating covariance, correlation, or cross product, the number of bytes of memory required is at least $(P^2P^T + P^T)b$, where $P$ is the dimension of the task or number of variables, $T$ is the number of threads, and $b$ is the number of bytes required for each unit of data. If observation is weighted and the method is \texttt{VSL_SS_METHOD_FAST}, then the memory required is at least $(P^2P^T + P^T + N^T)b$, where $N$ is the number of observations.

Summary Statistics Task Destructor

Task destructor is the \texttt{vslSSDeleteTask} routine intended to delete task objects and release memory.

\texttt{vslSSDeleteTask}

\textit{Destroys the task object and releases the memory.}

\textbf{Syntax}

\begin{verbatim}
status = vslSSDeleteTask(&task);
\end{verbatim}

\textbf{Include Files}

- \texttt{mkl.h}

\textbf{Input Parameters}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{task}</td>
<td>\texttt{VSLSSTaskPtr*}</td>
<td>Descriptor of the task to destroy</td>
</tr>
</tbody>
</table>

\textbf{Output Parameters}

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{status}</td>
<td>\texttt{int}</td>
<td>Sets to \texttt{VSL_STATUS_OK} if the task is deleted; otherwise a non-zero code is returned.</td>
</tr>
</tbody>
</table>

\textbf{Description}

The \texttt{vslSSDeleteTask} routine deletes the task descriptor object, releases the memory allocated for the structure, and sets the task pointer to \texttt{NULL}. If \texttt{vslSSDeleteTask} fails to delete the task successfully, it returns an error code.

\textbf{NOTE}

Call of the destructor with the \texttt{NULL} pointer as the parameter results in termination of the function with no system crash.
Summary Statistics Usage Examples

The following examples show various standard operations with Summary Statistics routines.

Calculating Fixed Estimates for Fixed Data

The example shows recurrent calculation of the same estimates with a given set of variables for the complete life cycle of the task in the case of a variance-covariance matrix. The set of vector components to process remains unchanged, and the data comes in blocks. Before you call the `vslSSCompute` routine, initialize pointers to arrays for mean and covariance and set buffers.

```c
... double w[2];
double indices[DIM] = {1, 0, 1};
/* calculating mean for 1st and 3rd random vector components */
/* Initialize parameters of the task */
p = DIM;
n = N;
xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;
w[0] = 0.0; w[1] = 0.0;
for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;
status = vsldSSNewTask( &task, &p, &n, &xstorage, x, 0, indices );
status = vsldSSEditTask ( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );
```

You can process data arrays that come in blocks as follows:

```c
for ( i = 0; i < num_of_blocks; i++ )
{}
status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );
/* Read new data block into array x */
```

Calculating Different Estimates for Variable Data

The context of your calculation may change in the process of data analysis. The example below shows the data that comes in two blocks. You need to estimate a covariance matrix for the complete data, and the third central moment for the second block of the data using the weights that were accumulated for the previous datasets. The second block of the data is stored in another array. You can proceed as follows:

```c
/* Set parameters for the task */
p = DIM;
n = N;
xstorage = VSL_SS_MATRIX_STORAGE_ROWS;
covstorage = VSL_SS_MATRIX_STORAGE_FULL;
w[0] = 0.0; w[1] = 0.0;
```
for ( i = 0; i < p; i++ ) mean[i] = 0.0;
for ( i = 0; i < p*p; i++ ) cov[i] = 0.0;

/* Create task */
status = vsldSSNewTask( &task, &p, &n, &xstorage, x1, 0, indices );

/* Initialize the task parameters */
status = vsldSSEditTask( task, VSL_SS_ED_ACCUM_WEIGHT, w );
status = vsldSSEditCovCor( task, mean, cov, &covstorage, 0, 0 );

/* Calculate covariance for the x1 data */
status = vsldSSCompute( task, VSL_SS_COV, VSL_SS_METHOD_FAST );

/* Initialize array of the 3d central moments and pass the pointer to the task */
for ( i = 0; i < p; i++ ) c3_m[i] = 0.0;

/* Modify task context */
status = vsldSSEditTask( task, VSL_SS_ED_3C_MOM, c3_m );
status = vsldSSEditTask( task, VSL_SS_ED_OBSERV, x2 );

/* Calculate covariance for the x1 & x2 data block */
/* Calculate the 3d central moment for the 2nd data block using earlier accumulated weight */
status = vsldSSCompute(task, VSL_SS_COV|VSL_SS_3C_MOM, VSL_SS_METHOD_FAST );
...
status = vsllSSDeleteTask( &task );

Similarly, you can modify indices of the variables to be processed for the next data block.

Summary Statistics Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel® MKL Summary Statistics functions.

Matrix and Weights of Observations

For a random $p$-dimensional vector $\xi = (\xi_1, ..., \xi_i, ..., \xi_p)$, this manual denotes the following:

- $(X)_i = (x_{ij})_{j=1..n}$ is the result of $n$ independent observations for the $i$-th component $\xi_i$ of the vector $\xi$.
- The two-dimensional array $X = (x_{ij})_{n \times p}$ is the matrix of observations.
- The column $[X]_j = (x_{ij})_{i=1..p}$ of the matrix $X$ is the $j$-th observation of the random vector $\xi$.

Each observation $[X]_j$ is assigned a non-negative weight $w_j$, where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to $n$ observations of the random vector $\xi$.
- $W = \sum_{i=1}^{n} w_i$

  is the accumulated weight corresponding to observations $X$.

Vector of sample means

$M(X) = (M_1(X), ..., M_p(X))$ with $M_i(X) = \frac{1}{n} \sum_{j=1}^{n} w_j x_{ij}$

for all $i = 1, ..., p$. 

Summary Statistics Mathematical Notation and Definitions

The following notations are used in the mathematical definitions and the description of the Intel® MKL Summary Statistics functions.

Matrix and Weights of Observations

For a random $p$-dimensional vector $\xi = (\xi_1, ..., \xi_i, ..., \xi_p)$, this manual denotes the following:

- $(X)_i = (x_{ij})_{j=1..n}$ is the result of $n$ independent observations for the $i$-th component $\xi_i$ of the vector $\xi$.
- The two-dimensional array $X = (x_{ij})_{n \times p}$ is the matrix of observations.
- The column $[X]_j = (x_{ij})_{i=1..p}$ of the matrix $X$ is the $j$-th observation of the random vector $\xi$.

Each observation $[X]_j$ is assigned a non-negative weight $w_j$, where

- The vector $(w_j)_{j=1..n}$ is a vector of weights corresponding to $n$ observations of the random vector $\xi$.
- $W = \sum_{i=1}^{n} w_i$

  is the accumulated weight corresponding to observations $X$.

Vector of sample means

$M(X) = (M_1(X), ..., M_p(X))$ with $M_i(X) = \frac{1}{n} \sum_{j=1}^{n} w_j x_{ij}$

for all $i = 1, ..., p$. 

Vector of sample partial sums

\[ S(X) = \left\{ S_1(X), \ldots, S_p(X) \right\} \] with \[ S_i(X) = \sum_{j=1}^{n} w_j x_{ij} \]
for all \( i = 1, \ldots, p. \)

Vector of sample variances

\[ V(X) = \left\{ V_1(X), \ldots, V_p(X) \right\} \] with \[ V_i(X) = \frac{1}{B} \sum_{j=1}^{n} w_j \left( x_{ij} - M_i(X) \right)^2, \]
\[ B = W - \sum_{j=1}^{n} w_j^2 / W \]
for all \( i = 1, \ldots, p. \)

Vector of sample raw/algebraic moments of \( k \)-th order, \( k \geq 1 \)

\[ R^{(k)}(X) = \left\{ R_1^{(k)}(X), \ldots, R_p^{(k)}(X) \right\} \] with \[ R_i^{(k)}(X) = \frac{1}{W} \sum_{j=1}^{n} w_j x_{ij}^k \]
for all \( i = 1, \ldots, p. \)

Vector of sample raw/algebraic partial sums of \( k \)-th order, \( k = 2, 3, 4 \) (raw/algebraic partial sums of squares/cubes/fourth powers)

\[ S^k(X) = \left\{ S_1^k(X), \ldots, S_p^k(X) \right\} \] with \[ S_i^k(X) = \sum_{j=1}^{n} w_j x_{ij}^k \]
for all \( i = 1, \ldots, p. \)

Vector of sample central moments of the third and the fourth order

\[ C^{(k)}(X) = \left\{ C_1^{(k)}(X), \ldots, C_p^{(k)}(X) \right\} \] with \[ C_i^{(k)}(X) = \frac{1}{B} \sum_{j=1}^{n} w_j \left( x_{ij} - M_i(X) \right)^k, \]
\[ B = \sum_{j=1}^{n} w_j \]
for all \( i = 1, \ldots, p \) and \( k = 3, 4. \)

Vector of sample central partial sums of \( k \)-th order, \( k = 2, 3, 4 \) (central partial sums of squares/cubes/fourth powers)

\[ S^k(X) = \left\{ S_1^k(X), \ldots, S_p^k(X) \right\} \] with \[ S_i^k(X) = \sum_{j=1}^{n} w_j \left( x_{ij} - S_i(X) \right)^k \]
for all \( i = 1, \ldots, p. \)

Vector of sample excess kurtosis values

\[ B(X) = \left\{ B_1(X), \ldots, B_p(X) \right\} \] with \[ B_i(X) = \frac{C_i^{(4)}(X)}{V_i^{(2)}(X)} - 3 \]
for all \( i = 1, \ldots, p. \)

Vector of sample skewness values

\[ \Gamma(X) = \left\{ \Gamma_1(X), \ldots, \Gamma_p(X) \right\} \] with \[ \Gamma_i(X) = \frac{C_i^{(3)}(X)}{V_i^{(2.5)}(X)} \]
for all \( i = 1, \ldots, p \).

**Vector of sample variation coefficients**

\[
VC(X) = \left( VC_1(X), \ldots, VC_p(X) \right) \quad \text{with} \quad VC_i(X) = \frac{V_i^{0.5}(X)}{M_i(X)}
\]

for all \( i = 1, \ldots, p \).

**Matrix of order statistics**

Matrix \( Y = (y_{i,j})_{p \times n} \) in which the \( i \)-th row \( (Y)_i = (y_{i,j})_{j=1,n} \) is obtained as a result of sorting in the ascending order of row \( (X)_i = (x_{i,j})_{j=1,n} \) in the original matrix of observations.

**Vector of sample minimum values**

\[
Min(X) = \left( Min_1(X), \ldots, Min_p(X) \right), \quad \text{where} \quad Min_i(X) = y_{i1}
\]

for all \( i = 1, \ldots, p \).

**Vector of sample maximum values**

\[
Max(X) = \left( Max_1(X), \ldots, Max_p(X) \right), \quad \text{where} \quad Max_i(X) = y_{in}
\]

for all \( i = 1, \ldots, p \).

**Vector of sample median values**

\[
Med(X) = \left( Med_1(X), \ldots, Med_p(X) \right), \quad \text{where} \quad Med_i(X) = \begin{cases} y_{i,(n+1)/2} & \text{if} \; n \text{ is odd} \\ \frac{y_{i,n/2} + y_{i,n/2+1}}{2} & \text{if} \; n \text{ is even} \end{cases}
\]

for all \( i = 1, \ldots, p \).

**Vector of sample median absolute deviations**

\[
MDAD(X) = \left( MDAD_1(X), \ldots, MDAD_p(X) \right), \quad \text{where} \quad MDAD_i(X) = Med_i(Z) \quad \text{with} \quad Z = \left( z_{ij} \right)_{i=1,p,j=1,n'}
\]

\[
z_{ij} = \left| x_{ij} - Med_i(X) \right|
\]

for all \( i = 1, \ldots, p \).

**Vector of sample mean absolute deviations**

\[
MNAD(X) = \left( MNAD_1(X), \ldots, MNAD_p(X) \right), \quad \text{where} \quad MNAD_i(X) = M_i(Z) \quad \text{with} \quad Z = \left( z_{ij} \right)_{i=1,p,j=1,n'}
\]

\[
z_{ij} = \left| x_{ij} - M_i(X) \right|
\]

for all \( i = 1, \ldots, p \).

**Vector of sample quantile values**

For a positive integer number \( q \) and \( k \) belonging to the interval \([0, q-1]\), point \( z_i \) is the \( k \)-th \( q \) quantile of the random variable \( \xi_i \) if \( P(\xi_i \leq z_i) \geq \beta \) and \( P(\xi_i \leq z_i) \geq 1 - \beta \), where

- \( P \) is the probability measure.
- \( \beta = k/n \) is the quantile order.

The calculation of quantiles is as follows:
\[ j = \lfloor(n-1)\beta\rfloor \text{ and } f = \{ (n-1)\beta \} \] as integer and fractional parts of the number \((n-1)\beta\), respectively, and the vector of sample quantile values is 
\[ Q(X, \beta) = \{Q_1(X, \beta), \ldots, Q_p(X, \beta)\} \]
where 
\[ (Q_i(X, \beta)) = y_{i,j+1} + f(y_{i,j+2} - y_{i,j+1}) \]
for all \(i = 1, \ldots, p\).

**Variance-covariance matrix**
\[ C(X) = \{c_{ij}(X)\} \_p \times p \]
where
\[ c_{ij}(X) = \frac{1}{B} \sum_{k=1}^{n} w_k(x_{ik} - M_j(X))(x_{jk} - M_j(X)), \quad B = W - \sum_{j=1}^{n} w_j^2 / W \]

**Cross-product matrix (matrix of cross-products and sums of squares)**
\[ CP(X) = \{cp_{ij}(X)\} \_p \times p \]
where
\[ cp_{ij}(X) = \sum_{k=1}^{n} w_k(x_{ik} - M_j(X))(x_{jk} - M_j(X)) \]

**Pooled and group variance-covariance matrices**
The set \(N = \{1, \ldots, n\}\) is partitioned into non-intersecting subsets
\[ G_i : i = 1, \ldots, g, \quad N = \bigcup_{i=1}^{g} G_i \]
The observation \(X_j = (x_{ij})_{i=1,\ldots,n}\) belongs to the group \(r\) if \(j \in G_r\). One observation belongs to one group only. The group mean and variance-covariance matrices are calculated similarly to the formulas above:
\[ M^{(r)}(X) = \{M_1^{(r)}(X), \ldots, M_p^{(r)}(X)\} \text{ with } M_i^{(r)}(X) = \frac{1}{W^{(r)}} \sum_{j \in G_r} w_j x_{ij} W^{(r)} = \sum_{j \in G_r} w_j \]
for all \(i = 1, \ldots, p\),
\[ C^{(r)}(X) = \{c_{ij}^{(r)}(X)\} \_p \times p \]
where
\[ c_{ij}^{(r)}(X) = \frac{1}{B^{(r)}} \sum_{k \in G_r} w_k(x_{ik} - M_i^{(r)}(X))(x_{jk} - M_j^{(r)}(X)), \quad B^{(r)} = W^{(r)} - \sum_{j \in G_r} w_j^2 / W^{(r)} \]
for all \(i = 1, \ldots, p\) and \(j = 1, \ldots, p\).

A pooled variance-covariance matrix and a pooled mean are computed as weighted mean over group covariance matrices and group means, correspondingly:
\[ M^{\text{pooled}}(X) = \{M_1^{\text{pooled}}(X), \ldots, M_p^{\text{pooled}}(X)\} \text{ with } M_i^{\text{pooled}}(X) = \frac{1}{W^{(1)} + \ldots + W^{(g)}} \sum_{r=1}^{g} W^{(r)} M_i^{(r)}(X) \]
for all \(i = 1, \ldots, p\).
\[ C_{pooled}(X) = (c_{ij}^{pooled}(X))_{p \times p}, \quad c_{ij}^{pooled}(X) = \frac{1}{B^{(1)} + \ldots + B^{(g)}} \sum_{r=1}^{g} B^{(r)} c_{ij}^{(r)}(X) \]

for all \( i = 1, \ldots, p \) and \( j = 1, \ldots, p \).

**Correlation matrix**

\[ R(X) = \left( r_{ij}(X) \right)_{p \times p}, \quad \text{where} \quad r_{ij}(X) = \frac{c_{ij}}{\sqrt{c_{ii} c_{jj}}} \]

for all \( i = 1, \ldots, p \) and \( j = 1, \ldots, p \).

**Partial variance-covariance matrix**

For a random vector \( \xi \) partitioned into two components \( Z \) and \( Y \), a variance-covariance matrix \( C \) describes the structure of dependencies in the vector \( \xi \):

\[ C(X) = \begin{pmatrix} C_Z(X) & C_{ZY}(X) \\ C_{YZ}(X) & C_Y(X) \end{pmatrix}. \]

The partial covariance matrix \( P(X) = (p_{ij}(X))_{k \times k} \) is defined as

\[ P(X) = C_Y(X) - C_{YZ}(X) C_Z^{-1} C_{ZY}(X), \]

where \( k \) is the dimension of \( Y \).

**Partial correlation matrix**

The following is a partial correlation matrix for all \( i = 1, \ldots, k \) and \( j = 1, \ldots, k \):

\[ RP(X) = \left( r_{ij}(X) \right)_{k \times k}, \quad \text{where} \quad r_{ij}(X) = \frac{p_{ij}(X)}{\sqrt{p_{ii}(X)p_{jj}(X)}} \]

where

- \( k \) is the dimension of \( Y \).
- \( p_{ij}(X) \) are elements of the partial variance-covariance matrix.

**Sorted dataset**

Matrix \( Y = (y_{ij})_{p \times n} \), in which the \( i \)-th row \( (Y)_i \) is obtained as a result of sorting in ascending order the row \( (X)_i = (x_{ij})_{1 \ldots n} \) in the original matrix of observations.
Fourier Transform Functions

The general form of the discrete Fourier transform is

\[ z_{k_1, k_2, \ldots, k_d} = \sigma \times \sum_{j_d = 0}^{n_d-1} \cdots \sum_{j_2 = 0}^{n_2-1} \sum_{j_1 = 0}^{n_1-1} w_{j_1, j_2, \ldots, j_d} \exp \left( \delta 2\pi \sum_{l=1}^{d} \frac{j_l k_l}{n_l} \right) \]

for \( k_1 = 0, \ldots, n_1-1 \) (\( l = 1, \ldots, d \)), where \( \sigma \) is a scale factor, \( \delta = -1 \) for the forward transform, and \( \delta = +1 \) for the inverse (backward) transform. In the forward transform, the input (periodic) sequence \( \{w_{j_1, j_2, \ldots, j_d}\} \) belongs to the set of complex-valued sequences and real-valued sequences. Respective domains for the backward transform are represented by complex-valued sequences and complex-valued conjugate-even sequences.

The Intel® Math Kernel Library (Intel® MKL) provides an interface for computing a discrete Fourier transform through the fast Fourier transform algorithm. Prefixes Dfti in function names and DFTI in the names of configuration parameters stand for Discrete Fourier Transform Interface.

This chapter describes the following implementations of the fast Fourier transform functions available in Intel® MKL:

- Fast Fourier transform (FFT) functions for single-processor or shared-memory systems (see FFT Functions)
- Cluster FFT functions for distributed-memory architectures (available only for Intel® 64 and Intel® Many Integrated Core architectures)

**NOTE**
Intel® MKL also supports the FFTW3* interfaces to the fast Fourier transform functionality for shared memory paradigm (SMP) systems.

Both FFT and Cluster FFT functions compute an FFT in five steps:

1. Allocate a fresh descriptor for the problem with a call to the DftiCreateDescriptor or DftiCreateDescriptorDM function. The descriptor captures the configuration of the transform, such as the dimensionality (or rank), sizes, number of transforms, memory layout of the input/output data (defined by strides), and scaling factors. Many of the configuration settings are assigned default values in this call which you might need to modify in your application.

2. Optionally adjust the descriptor configuration with a call to the DftiSetValue or DftiSetValueDM function as needed. Typically, you must carefully define the data storage layout for an FFT or the data distribution among processes for a Cluster FFT. The configuration settings of the descriptor, such as the default values, can be obtained with the DftiGetValue or DftiGetValueDM function.

3. Commit the descriptor with a call to the DftiCommitDescriptor or DftiCommitDescriptorDM function, that is, make the descriptor ready for the transform computation. Once the descriptor is committed, the parameters of the transform, such as the type and number of transforms, strides and distances, the type and storage layout of the data, and so on, are "frozen" in the descriptor.

4. Compute the transform with a call to the DftiComputeForward/DftiComputeBackward or DftiComputeForwardDM/DftiComputeBackwardDM functions as many times as needed. Because the descriptor is defined and committed separately, all that the compute functions do is take the input and output data and compute the transform as defined. To modify any configuration parameters for another call to a compute function, use DftiSetValue followed by DftiCommitDescriptor (DftiSetValueDM followed by DftiCommitDescriptorDM) or create and commit another descriptor.

5. Deallocate the descriptor with a call to the DftiFreeDescriptor or DftiFreeDescriptorDM function. This returns the memory internally consumed by the descriptor to the operating system.
All the above functions return an integer status value, which is zero upon successful completion of the operation. You can interpret a non-zero status with the help of the `DftiErrorClass` or `DftiErrorMessage` function.

The FFT functions support lengths with arbitrary factors. You can improve performance of the Intel® MKL FFT if the length of your data vector permits factorization into powers of optimized radices. See the Intel® MKL Developer Guide for specific radices supported efficiently.

**NOTE**
The FFT functions assume the Cartesian representation of complex data (that is, the real and imaginary parts define a complex number). The Intel® MKL Vector Mathematical Functions provide efficient tools for conversion to and from polar representation (see Example "Conversion from Cartesian to polar representation of complex data" and Example "Conversion from polar to Cartesian representation of complex data").

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### FFT Functions

The fast Fourier transform function library of Intel® MKL provides one-dimensional, two-dimensional, and multi-dimensional transforms (of up to seven dimensions) and offers both Fortran and C interfaces for all transform functions.

Table "FFT Functions in Intel® MKL" lists FFT functions implemented in Intel® MKL:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Descriptor Manipulation Functions</strong></td>
<td></td>
</tr>
<tr>
<td><code>DftiCreateDescriptor</code></td>
<td>Allocates the descriptor data structure and initializes it with default configuration values.</td>
</tr>
<tr>
<td><code>DftiCommitDescriptor</code></td>
<td>Performs all initialization for the actual FFT computation.</td>
</tr>
<tr>
<td><code>DftiFreeDescriptor</code></td>
<td>Frees memory allocated for a descriptor.</td>
</tr>
<tr>
<td><code>DftiCopyDescriptor</code></td>
<td>Makes a copy of an existing descriptor.</td>
</tr>
<tr>
<td><strong>FFT Computation Functions</strong></td>
<td></td>
</tr>
<tr>
<td><code>DftiComputeForward</code></td>
<td>Computes the forward FFT.</td>
</tr>
<tr>
<td><code>DftiComputeBackward</code></td>
<td>Computes the backward FFT.</td>
</tr>
<tr>
<td><strong>Descriptor Configuration Functions</strong></td>
<td></td>
</tr>
<tr>
<td><code>DftiSetValue</code></td>
<td>Sets one particular configuration parameter with the specified configuration value.</td>
</tr>
</tbody>
</table>
### Function Name | Operation
--- | ---
DftiGetValue | Gets the value of one particular configuration parameter.

**Status Checking Functions**

DftiErrorClass | Checks if the status reflects an error of a predefined class.
DftiErrorMessage | Translates the numeric value of an error status into a message.

### FFT Interface

The Intel® MKL FFT functions are provided with the Fortran and C interfaces.

The materials presented in this chapter assume the availability of native complex types in C as they are specified in C9X.

To use the FFT functions, you need to include `mkl_dfti.h` in your C code.

The C interface provides the `DFTI_DESCRIPTOR_HANDLE` type, named constants of two enumeration types `DFTI_CONFIG_PARAM` and `DFTI_CONFIG_VALUE`, and functions, some of which accept different numbers of input arguments.

**NOTE**
The current version of the library may not support some of the FFT functions or functionality described in the subsequent sections of this chapter. You can find the complete list of the implementation-specific exceptions in the Intel® MKL Release Notes.

For the main categories of Intel® MKL FFT functions, see [FFT Functions](#).

### Computing an FFT

You can find code examples that compute transforms in the [Fourier Transform Functions Code Examples](#) section in the Code Examples appendix.

Usually you can compute an FFT by five function calls (refer to the usage model for details). A single data structure, the descriptor, stores configuration parameters that can be changed independently.

The descriptor data structure, when created, contains information about the length and domain of the FFT to be computed, as well as the setting of several configuration parameters. Default settings for some of these parameters are as follows:

- Scale factor: none
- Number of data sets: one
- Data storage: contiguous
- Placement of results: in-place (the computed result overwrites the input data)

The default settings can be changed one at a time through the function `DftiSetValue` as illustrated in Example "Changing Default Settings (C)".

### Configuration Settings

Each of the configuration parameters is identified by a named constant in the `MKL_DFTI` module. These named constants have the enumeration type `DFTI_CONFIG_PARAM`.

All the Intel® MKL FFT configuration parameters are readable. Some of them are read-only, while others can be set using the `DftiCreateDescriptor` or `DftiSetValue` function.

Values of the configuration parameters fall into the following groups:
Values that have native data types. For example, the number of simultaneous transforms requested has an integer value, while the scale factor for a forward transform is a single-precision number.

Values that are discrete in nature and are provided in the MKL_DFTI module as named constants. For example, the domain of the forward transform requires values to be named constants. The named constants for configuration values have the enumeration type DFTI_CONFIG_VALUE.

The Table "Configuration Parameters" summarizes the information on configuration parameters, along with their types and values. For more details of each configuration parameter, see the subsection describing this parameter.

**Configuration Parameters**

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Type/Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_PRECISION</td>
<td>Named constant</td>
<td>Precision of the computation.</td>
</tr>
<tr>
<td>DFTI_FORWARD_DOMAIN</td>
<td>Named constant</td>
<td>Type of the transform.</td>
</tr>
<tr>
<td>DFTI_DIMENSION</td>
<td>Integer scalar</td>
<td>Dimension of the transform.</td>
</tr>
<tr>
<td>DFTI_LENGTHS</td>
<td>Integer scalar/array</td>
<td>Lengths of each dimension.</td>
</tr>
</tbody>
</table>

**Most common configuration parameters, no default, must be set explicitly by DftiCreateDescriptor**

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION
- DFTI_LENGTHS

**Common configuration parameters, settable by DftiSetValue**

- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE
- DFTI_BACKWARD_SCALE
- DFTI_NUMBER_OF_USER_THREADS
- DFTI_THREAD_LIMIT
- DFTI_DESCRIPTOR_NAME

**Data layout configuration parameters for single and multiple transforms. Settable by DftiSetValue**

- DFTI_INPUT_STRIDES

Precision of the value should be the same as defined by DFTI_PRECISION.
<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Type/Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_OUTPUT_STRIDES</td>
<td>Integer array</td>
<td>Defines the output data layout.</td>
</tr>
<tr>
<td>DFTI_NUMBER_OF_TRANSFORMS</td>
<td>Integer scalar</td>
<td>Number of transforms. Default value: 1.</td>
</tr>
<tr>
<td>DFTI_INPUT_DISTANCE</td>
<td>Integer scalar</td>
<td>Defines the distance between input data sets for multiple transforms.</td>
</tr>
<tr>
<td>DFTI_OUTPUT_DISTANCE</td>
<td>Integer scalar</td>
<td>Defines the distance between output data sets for multiple transforms.</td>
</tr>
<tr>
<td>DFTI_COMPLEX_STORAGE</td>
<td>Named constant</td>
<td>Defines whether the real and imaginary parts of data for a complex transform are interleaved in one array or split in two arrays. Default value: DFTI_COMPLEX_COMPLEX.</td>
</tr>
<tr>
<td>DFTI_REAL_STORAGE</td>
<td>Named constant</td>
<td>Defines how real data for a real transform is stored. Only the DFTI_REAL_REAL value is supported.</td>
</tr>
<tr>
<td>DFTI_CONJUGATE_EVEN_STORAGE</td>
<td>Named constant</td>
<td>Defines whether the complex data in the backward domain of a real transform is stored as complex elements or as real elements. For the default value, see the detailed description.</td>
</tr>
<tr>
<td>DFTI_Packed_FORMAT</td>
<td>Named constant</td>
<td>Defines the layout of real elements in the backward domain of a one-dimensional or two-dimensional real transform.</td>
</tr>
</tbody>
</table>

**Advanced configuration parameters, settable by** DftiSetValue

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Type/Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_WORKSPACE</td>
<td>Named constant</td>
<td>Defines whether the library should prefer algorithms using additional memory. Default value: DFTI_ALLOW.</td>
</tr>
<tr>
<td>DFTI_ORDERING</td>
<td>Named constant</td>
<td>Defines whether the result of a complex transform is ordered or permuted. Default value: DFTI_ORDERED.</td>
</tr>
</tbody>
</table>

**Read-Only configuration parameters**

<table>
<thead>
<tr>
<th>Configuration Parameter</th>
<th>Type/Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_COMMIT_STATUS</td>
<td>Named constant</td>
<td>Readiness of the descriptor for computation.</td>
</tr>
<tr>
<td>DFTI_VERSION</td>
<td>String</td>
<td>Version of Intel® MKL. Assumed length of the string is DFTI_VERSION_LENGTH.</td>
</tr>
</tbody>
</table>

**See Also**

Configuring and Computing an FFT in C/C++
**DFTI_PRECISION**

The configuration parameter `DFTI_PRECISION` denotes the floating-point precision in which the transform is to be carried out. A setting of `DFTI_SINGLE` stands for single precision, and a setting of `DFTI_DOUBLE` stands for double precision. The data must be presented in this precision, the computation is carried out in this precision, and the result is delivered in this precision.

`DFTI_PRECISION` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.

To better understand configuration of the precision of transforms, refer to these examples in your Intel® MKL directory:

`.examples/dftc/source/basic_sp_complex_dft_1d.c`

`.examples/dftc/source/basic_dp_complex_dft_1d.c`

**See Also**

`DFTI_FORWARD_DOMAIN`, `DFTI_DIMENSION`, `DFTI_LENGTHS`, `DftiCreateDescriptor`

**DFTI_FORWARD_DOMAIN**

The general form of a discrete Fourier transform is

\[
\begin{align*}
z_{k_1, k_2, \ldots, k_d} &= \sigma \times \sum_{j_d=0}^{n_d-1} \sum_{j_2=0}^{n_2-1} \sum_{j_1=0}^{n_1-1} w_{j_1, j_2, \ldots, j_d} \exp \left( \delta \frac{2\pi}{n_1} \sum_{l=1}^{d} j_l k_l / n_l \right)
\end{align*}
\]

for \( k_1 = 0, \ldots, n_1-1 \) (\( l = 1, \ldots, d \)), where \( \sigma \) is a scale factor, \( \delta = -1 \) for the forward transform, and \( \delta = +1 \) for the backward transform.

The Intel® MKL implementation of the FFT algorithm, used for fast computation of discrete Fourier transforms, supports forward transforms on input sequences of two domains, as specified by the `DFTI_FORWARD_DOMAIN` configuration parameter: general complex-valued sequences (`DFTI_COMPLEX` domain) and general real-valued sequences (`DFTI_REAL` domain). The forward transform maps the forward domain to the corresponding backward domain, as shown in Table "Correspondence of Forward and Backward Domain".

The conjugate-even domain covers complex-valued sequences with the symmetry property:

\[
x(k_1, k_2, \ldots, k_d) = \text{conjugate}(x(n_1-k_1, n_2-k_2, \ldots, n_d-k_d))
\]

where the index arithmetic is performed modulo respective size, that is,

\[
x(\ldots, expr_s, \ldots) \equiv x(\ldots, \text{mod}(expr_s, n_s), \ldots),
\]

and therefore

\[
x(\ldots, n_s, \ldots) \equiv x(\ldots, 0, \ldots).
\]

Due to this property of conjugate-even sequences, only a part of such sequence is stored in the computer memory, as described in `DFTI_CONJUGATE_EVEN_STORAGE`.

**Correspondence of Forward and Backward Domain**

<table>
<thead>
<tr>
<th>Forward Domain</th>
<th>Implied Backward Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex (DFTI_COMPLEX)</td>
<td>Complex (DFTI_COMPLEX)</td>
</tr>
<tr>
<td>Real (DFTI_REAL)</td>
<td>Conjugate-even</td>
</tr>
</tbody>
</table>

`DFTI_FORWARD_DOMAIN` does not have a default value. Set it explicitly by calling the `DftiCreateDescriptor` function.
To better understand usage of the `DFTI_FORWARD_DOMAIN` configuration parameter, you can refer to these examples in your Intel® MKL directory:

`./examples/dftc/source/basic_sp_complex_dft_1d.c`
`./examples/dftc/source/basic_sp_real_dft_1d.c`

See Also
- `DFTI_PRECISION`
- `DFTI_DIMENSION, DFTI_LENGTHS`
- `DftiCreateDescriptor`

**DFTI_DIMENSION, DFTI_LENGTHS**

The dimension of the transform is a positive integer value represented in an integer scalar of `MKL_LONG` data type. For a one-dimensional transform, the transform length is specified by a positive integer value represented in an integer scalar of `MKL_LONG` data type. For multi-dimensional (≥ 2) transform, the lengths of each of the dimensions are supplied in an integer array (of `MKL_LONG` data type).

`DFTI_DIMENSION` and `DFTI_LENGTHS` do not have a default value. To set them, use the `DftiCreateDescriptor` function and not the `DftiSetValue` function.

To better understand usage of the `DFTI_DIMENSION` and `DFTI_LENGTHS` configuration parameters, you can refer to basic examples of one-, two-, and three-dimensional transforms in your Intel® MKL directory. Naming conventions for the examples are self-explanatory. For example, refer to these examples of single-precision two-dimensional transforms:

`./examples/dftc/source/basic_sp_real_dft_2d.c`
`./examples/dftc/source/basic_sp_complex_dft_2d.c`

See Also
- `DFTI_FORWARD_DOMAIN`
- `DFTI_PRECISION`
- `DftiCreateDescriptor`
- `DftiSetValue`

**DFTI_PLACEMENT**

By default, the computational functions overwrite the input data with the output result. That is, the default setting of the configuration parameter `DFTI_PLACEMENT` is `DFTI_INPLACE`. You can change that by setting it to `DFTI_NOT_INPLACE`.

**NOTE**

When the configuration parameter is set to `DFTI_NOT_INPLACE`, the input and output data sets must have no common elements.

To better understand the use of the `DFTI_PLACEMENT` configuration parameter, see this example in your Intel® MKL directory:

`./examples/dftc/source/config_placement.c`

See Also
- `DftiSetValue`
**DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE**

The forward transform and backward transform are each associated with a scale factor $\sigma$ of its own having the default value of 1. You can specify the scale factors using one or both of the configuration parameters DFTI_FORWARD_SCALE and DFTI_BACKWARD_SCALE. For example, for a one-dimensional transform of length $n$, you can use the default scale of 1 for the forward transform and set the scale factor for the backward transform to be $1/n$, thus making the backward transform the inverse of the forward transform.

Set the scale factor configuration parameter using a real floating-point data type of the same precision as the value for DFTI_PRECISION.

**NOTE**

For inquiry of the scale factor with the DftiGetValue function, the `config_val` parameter must have the same floating-point precision as the descriptor.

**See Also**

DftiSetValue
DFTI_PRECISION
DftiGetValue

**DFTI_NUMBER_OF_USER_THREADS**

The DFTI_NUMBER_OF_USER_THREADS configuration parameter is no longer used and kept for compatibility with previous versions of Intel® MKL.

**See Also**

DftiSetValue

**DFTI_THREAD_LIMIT**

In some situations you may need to limit the number of threads that the DftiComputeForward and DftiComputeBackward functions use. For example, if more than one thread calls Intel® MKL, it might be important that the thread calling these functions does not oversubscribe computing resources (CPU cores). Similarly, a known limit of the maximum number of threads to be used in computations might help the DftiCommitDescriptor function to select a more optimal computation method.

Set the parameter DFTI_THREAD_LIMIT as follows:

- To a positive number, to specify the maximum number of threads to be used by the compute functions.
- To zero (the default value), to use the maximum number of threads permitted in Intel® MKL FFT functions.

See "Techniques to Set the Number of Threads" in the Intel® MKL Developer Guide for more information.

On an attempt to set a negative value, the DftiSetValue function returns an error and does not update the descriptor.

The value of the DFTI_THREAD_LIMIT configuration parameter returned by the DftiGetValue function is defined as follows:

<table>
<thead>
<tr>
<th>Commit Status</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not committed</td>
<td>The value of DFTI_THREAD_LIMIT set in a previous call to the DftiSetValue function or the default value</td>
</tr>
<tr>
<td>Committed</td>
<td>The upper limit on the number of threads used by the DftiComputeForward and DftiComputeBackward functions</td>
</tr>
</tbody>
</table>
To better understand the use of the DFTI_THREAD_LIMIT configuration parameter, see this example in your Intel® MKL directory:

```
./examples/dftc/source/config_thread_limit.c
```

**See Also**

DftiGetValue
DftiSetValue
DftiCommitDescriptor
DftiComputeForward
DftiComputeBackward

**Threading Control Functions**

**DFTI_COMMIT_STATUS**

**DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES**

The FFT interface provides configuration parameters that define the layout of multidimensional data in the computer memory. For \( d \)-dimensional data set \( X \) defined by dimensions \( N_1 \times N_2 \times \ldots \times N_d \), the layout describes where a particular element \( X(k_1, k_2, \ldots, k_d) \) of the data set is located. The memory address of the element \( X(k_1, k_2, \ldots, k_d) \) is expressed by the formula,

\[
\text{address of } X(k_1, k_2, \ldots, k_d) = \text{the pointer supplied to the compute function} + s_0 + k_1s_1 + k_2s_2 + \ldots + k_ds_d,
\]

where \( s_0 \) is the displacement and \( s_1, \ldots, s_d \) are generalized strides. The configuration parameters DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES enable you to get and set these values. The configuration value is an array of values \( (s_0, s_1, \ldots, s_d) \) of MKL_LONG data type.

The offset is counted in elements of the data type (complex or real) defined by the descriptor configuration as tabulated below. The computation functions take pointers which are typed as real arrays regardless of the descriptor configuration. Specifically, the DFTI_FORWARD_DOMAIN, DFTI_COMPLEX_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the type of the elements as shown in Table "Assumed Element Types of the Input/Output Data":

<table>
<thead>
<tr>
<th>Descriptor Configuration</th>
<th>Element Type in the Forward Domain</th>
<th>Element Type in the Backward Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_COMPLEX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFTI_FORWARD_DOMAIN=DFTI_COMPLEX</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>DFTI_COMPLEX_STORAGE=DFTI_REAL_REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFTI_FORWARD_DOMAIN=DFTI_REAL</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>DFTI_COMPLEX_STORAGE=DFTI_COMPLEX_REAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DFTI_FORWARD_DOMAIN=DFTI_REAL</td>
<td>Real</td>
<td>Complex</td>
</tr>
<tr>
<td>DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The DFTI_INPUT_STRIDES configuration parameter defines the layout of the input data, while the element type is defined by the forward domain for the DftiComputeForward function and by the backward domain for the DftiComputeBackward function. The DFTI_OUTPUT_STRIDES configuration parameter defines the layout of the output data, while the element type is defined by the backward domain for the DftiComputeForward function and by the forward domain for DftiComputeBackward function.
NOTE
The DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES configuration parameters define the layout of input and output data, and not the forward-domain and backward-domain data. If the data layouts in forward domain and backward domain differ, set DFTI_INPUT_STRIDES and DFTI_OUTPUT_STRIDES explicitly and then commit the descriptor before calling computation functions.

For in-place transforms (DFTI_PLACEMENT=DFTI_INPLACE), the configuration set by DFTI_OUTPUT_STRIDES is ignored when the element types in the forward and backward domains are the same. If they are different, set DFTI_OUTPUT_STRIDES explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the first elements on input and output must coincide in each dimension.

The FFT interface supports both positive and negative stride values. If you use negative strides, set the displacement of the data as follows:

\[ s_0 = \sum_{i=1}^{d} (N_i - 1) \cdot \max(-s_i, 0). \]

The default setting of strides in a general multi-dimensional case assumes that the array that contains the data has no padding. The order of the strides depends on the programming language. For example:

```c
MKL_LONG dims[] = { n_d, …, n_2, n_1 };
DftiCreateDescriptor( &hand, precision, domain, d, dims );
// The above call assumes data declaration: type X[n_d]…[n_2][n_1]
// Default strides are { 0, n_d-1*…*n_2*n_1, …, n_2*n_1*1, n_1*1, 1 }
```

Note that in case of a real FFT (DFTI_FORWARD_DOMAIN=DFTI_REAL), where different data layouts in the backward domain are available (see DFTI_PACKED_FORMAT), the default value of the strides is not intuitive for the recommended CCE format (configuration setting DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX). In case of an in-place real transform with the CCE format, set the strides explicitly, as follows:

```c
MKL_LONG rstrides[] = { 0, 2*n_d-1*…*n_2*(n_1/2+1), …, 2*n_2*(n_1/2+1), 2*(n_1/2+1), 1 };
MKL_LONG cstrides[] = { 0, n_d-1*…*n_2*(n_1/2+1), …, n_2*(n_1/2+1), (n_1/2+1), 1 };
DftiCreateDescriptor( &hand, precision, DFTI_REAL, d, dims );
DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
// Set the strides appropriately for forward/backward transform
```

To better understand configuration of strides, you can also refer to these examples in your Intel® MKL directory:

- ./examples/dftc/source/basic_sp_complex_dft_2d.c
- ./examples/dftc/source/basic_sp_complex_dft_3d.c
- ./examples/dftc/source/basic_dp_complex_dft_2d.c
- ./examples/dftc/source/basic_dp_complex_dft_3d.c
- ./examples/dftc/source/basic_sp_real_dft_2d.c
- ./examples/dftc/source/basic_sp_real_dft_3d.c
- ./examples/dftc/source/basic_dp_real_dft_2d.c
- ./examples/dftc/source/basic_dp_real_dft_3d.c
- ./examples/dftc/source/basic_sp_real_dft_3d.c

See Also
DFTI_FORWARD_DOMAIN
DFTI_PLACEMENT
FFT Code Examples
If you need to perform a large number of identical FFTs, you can do this in a single call to a \texttt{DftiCompute*} function with the value of the \texttt{DFTI_NUMBER_OF_TRANSFORMS} configuration parameter equal to the actual number of the transforms. The default value of this parameter is one. You can set this parameter to a positive integer value using the \texttt{MKL_LONG} data type. When setting the number of transforms to a value greater than one, you also need to specify the distance between the input data sets and the distance between the output data sets using one of the \texttt{DFTI_INPUT_DISTANCE} and \texttt{DFTI_OUTPUT_DISTANCE} configuration parameters or both.

### Important
- The data sets to be transformed must not have common elements.
- All the sets of data must be located within the same memory block.

To better understand the use of the \texttt{DFTI_NUMBER_OF_TRANSFORMS} configuration parameter, see this example in your Intel® MKL directory:

\texttt{./examples/dftc/source/config_number_of_transforms.c}

### See Also
- FFT Computation Functions
- \texttt{DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE}
- \texttt{DftiSetValue}

### DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE

The FFT interface in Intel® MKL enables computation of multiple transforms. To compute multiple transforms, you need to specify the data distribution of the multiple sets of data. The distance between the first data elements of consecutive data sets, \texttt{DFTI_INPUT_DISTANCE} for input data or \texttt{DFTI_OUTPUT_DISTANCE} for output data, specifies the distribution. The configuration setting is a value of \texttt{MKL_LONG} data type.

The default value for both configuration settings is one. You must set this parameter explicitly if the number of transforms is greater than one (see \texttt{DFTI_NUMBER_OF_TRANSFORMS}).

The distance is counted in elements of the data type defined by the descriptor configuration (rather than by the type of the variable passed to the computation functions). Specifically, the \texttt{DFTI_FORWARD_DOMAIN}, \texttt{DFTI_COMPLEX_STORAGE}, and \texttt{DFTI_CONJUGATE_EVEN_STORAGE} configuration parameters define the type of the elements as shown in Table "Assumed Element Types of the Input/Output Data".

### NOTE
The configuration parameters \texttt{DFTI_INPUT_DISTANCE} and \texttt{DFTI_OUTPUT_DISTANCE} define the distance within input and output data, and not within the forward-domain and backward-domain data. If the distances in the forward and backward domains differ, set \texttt{DFTI_INPUT_DISTANCE} and \texttt{DFTI_OUTPUT_DISTANCE} explicitly and then commit the descriptor before calling computation functions.
For in-place transforms (\texttt{DFTI\_PLACEMENT=DFTI\_INPLACE}), the configuration set by \texttt{DFTI\_OUTPUT\_DISTANCE} is ignored when the element types in the forward and backward domains are the same. If they are different, set \texttt{DFTI\_OUTPUT\_DISTANCE} explicitly (even though the transform is in-place). Ensure a consistent configuration for in-place transforms, that is, the locations of the data sets on input and output must coincide.

This example illustrates setting of the \texttt{DFTI\_INPUT\_DISTANCE} configuration parameter:

```c
MKL\_LONG dims[] = { n1, \ldots, n2, n1 }; 
MKL\_LONG distance = nd\_\star n2\_\star n1; 
DftiCreateDescriptor( &hand, precision, DFTI\_COMPLEX, d, dims ); 
DftiSetValue( hand, DFTI\_NUMBER\_OF\_TRANSFORMS, (MKL\_LONG)howmany ); 
DftiSetValue( hand, DFTI\_INPUT\_DISTANCE, distance );
```

To better understand configuration of the distances, see these code examples in your Intel\textsuperscript{\textregistered} MKL directory:

```
./examples/dftc/source/config_number_of_transforms.c
```

See Also
---
\texttt{DFTI\_PLACEMENT}  
\texttt{DftiSetValue}  
\texttt{DftiCommitDescriptor}  
\texttt{DftiComputeForward}  
\texttt{DftiComputeBackward}  

**DFTI\_COMPLEX\_STORAGE, DFTI\_REAL\_STORAGE, DFTI\_CONJUGATE\_EVEN\_STORAGE**

Depending on the value of the \texttt{DFTI\_FORWARD\_DOMAIN} configuration parameter, the implementation of FFT supports several storage schemes for input and output data (see document [3] for the rationale behind the definition of the storage schemes). The data elements are placed within contiguous memory blocks, defined with generalized strides (see \texttt{DFTI\_INPUT\_STRIDES}, \texttt{DFTI\_OUTPUT\_STRIDES}). For multiple transforms, all sets of data should be located within the same memory block, and the data sets should be placed at the same distance from each other (see \texttt{DFTI\_NUMBER\_OF\_TRANSFORMS} and \texttt{DFTI\_INPUT\_DISTANCE}, \texttt{DFTI\_OUTPUT\_DISTANCE}).

**Tip**

Avoid setting up multidimensional arrays with lists of pointers to one-dimensional arrays. Instead use a one-dimensional array with the explicit indexing to access the data elements.

**FFT Examples** demonstrate the usage of storage formats.

**DFTI\_COMPLEX\_STORAGE: storage schemes for a complex domain**

For the \texttt{DFTI\_COMPLEX} forward domain, both input and output sequences belong to a complex domain. In this case, the configuration parameter \texttt{DFTI\_COMPLEX\_STORAGE} can have one of the two values: \texttt{DFTI\_COMPLEX\_COMPLEX} (default) or \texttt{DFTI\_REAL\_REAL}.

**NOTE**

In the Intel\textsuperscript{\textregistered} MKLFFT interface, storage schemes for a forward complex domain and the respective backward complex domain are the same.

With \texttt{DFTI\_COMPLEX\_COMPLEX} storage, complex-valued data sequences are referenced by a single complex parameter (array) \(A_Z\) so that a complex-valued element \(z_{k_1, k_2, \ldots, k_d}\) of the \(m\)-th \(d\)-dimensional sequence is located at \(A_Z[m\_\star \text{distance} + k_1\_\star \text{stride}_0 + k_2\_\star \text{stride}_1 + k_2\_\star \text{stride}_2 + \ldots + k_d\_\star \text{stride}_d]\) as a structure consisting of the real and imaginary parts.
This code illustrates the use of the `DFTI_COMPLEX_COMPLEX` storage:

```c
complex *AZ = malloc( N1*N2*N3*M * sizeof(AZ[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
// on input: Z{k1,k2,k3,m}
status = DftiComputeForward( desc, AZ );
// on output: Z{k1,k2,k3,m}
```

With the `DFTI_REAL_REAL` storage, complex-valued data sequences are referenced by two real parameters `AR` and `AI` so that a complex-valued element \( z_{k_1,k_2,\ldots,k_d} \) of the \( m \)-th sequence is computed as
\[
AR[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \ldots k_d*stride_d] + \sqrt{-1} * AI[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \ldots k_d*stride_d].
\]

This code illustrates the use of the `DFTI_REAL_REAL` storage:

```c
float *AR = malloc( N1*N2*N3*M * sizeof(AR[0]) );
float *AI = malloc( N1*N2*N3*M * sizeof(AI[0]) );
MKL_LONG ios[4], iodist; // input/output strides and distance
...
// on input: Z{k1,\ldots,kd,m}
status = DftiComputeForward( desc, AR, AI );
// on output: Z{k1,\ldots,kd,m}
```

**DFTI_REAL_STORAGE:** storage schemes for a real domain

The Intel® MKLFFT interface supports only one configuration value for this storage scheme: `DFTI_REAL_REAL`. With the `DFTI_REAL_REAL` storage, real-valued data sequences in a real domain are referenced by one real parameter `AR` so that real-valued element of the \( m \)-th sequence is located as
\[
AR[m*distance + stride_0 + k_1*stride_1 + k_2*stride_2 + \ldots k_d*stride_d].
\]

**DFTI_CONJUGATE_EVEN_STORAGE:** storage scheme for a conjugate-even domain

The Intel® MKLFFT interface supports two configuration values for this parameter: `DFTI_COMPLEX_REAL` (default) and `DFTI_COMPLEX_COMPLEX`. In both cases, the conjugate-even symmetry of the data enables storing only about a half of the whole mathematical result, so that one part of it can be directly referenced in the memory while the other part can be reconstructed depending on the selected storage configuration.

With the `DFTI_COMPLEX_REAL` storage, the complex-valued data sequences in the conjugate-even domain can be reconstructed as described in section **DFTI_PACKED_FORMAT**.

---

**Important**

Although `DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL` is the default setting for the `DFTI_REAL` forward domain, avoid using this storage scheme. This storage scheme is deprecated for one-dimensional transforms and the default will change in a future release. For two-dimensional transforms, this storage scheme is deprecated, support will be removed, and the default will change in a future release.

---

With the `DFTI_COMPLEX_COMPLEX` storage, the complex-valued data sequences in the conjugate-even domain are referenced by one complex parameter `AZ` so that a complex-valued element \( z_{k_1,k_2,\ldots,k_d} \) of the \( m \)-th sequence can be referenced or reconstructed as described below.
Important
Use the DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX configuration setting, which will become the default in future. This setting is supported for all transform ranks, provides a uniform pattern for reconstructing the entire conjugate-even sequence from the part of it that is actually stored, and is compatible with data layouts supported by other FFT libraries, such as FFTW. This storage scheme disregards the setting of DFTI_PACKED_FORMAT.

Consider a d-dimensional real-to-complex transform

\[ Z_{k_1, k_2, \ldots, k_d} = \sum_{n_1=0}^{N_1-1} \cdots \sum_{n_d=0}^{N_d-1} R_{n_1, n_2, \ldots, n_d} \frac{-2\pi i}{N_1} k_1 n_1 \cdots \frac{-2\pi i}{N_d} k_d n_d \]

Because the input sequence \( R \) is real-valued, the mathematical result \( Z \) has conjugate-even symmetry:

\[ z_{k_1, k_2, \ldots, k_d} = \text{conjugate}(z_{N_1-k_1, N_2-k_2, \ldots, N_d-k_d}), \]

where index arithmetic is performed modulo the length of the respective dimension. Obviously, the first element of the result is real-valued:

\[ z_{0, 0, \ldots, 0} = \text{conjugate}(z_{0, 0, \ldots, 0}). \]

For dimensions with even lengths, some of the other elements are real-valued too. For example, if \( N_e \) is even,

\[ z_{0, 0, \ldots, N_e/2, 0, \ldots, 0} = \text{conjugate}(z_{0, 0, \ldots, N_e/2, 0, \ldots, 0}). \]

With the conjugate-even symmetry, approximately a half of the result suffices to fully reconstruct it. For an arbitrary dimension \( h \), it suffices to store elements \( z_{k_1, k_2, \ldots, k_d} \) for the following indices:

- \( k_h = 0, \ldots, [N_h / 2] \)
- \( k_i = 0, \ldots, N_i - 1, \) where \( i = 1, \ldots, d \) and \( i \neq h \)

The symmetry property enables reconstructing the remaining elements: for \( k_h = [N_h / 2] + 1, \ldots, N_h - 1 \). In the Intel® MKLFFT interface, the halved dimension is the last dimension.

The following code illustrates usage of the DFTI_COMPLEX_COMPLEX storage for a conjugate-even domain:

```c
float *AR = malloc( N1*N2*M * sizeof(AR[0]) );
complex *AZ = malloc( N1*(N2/2+1)*M * sizeof(AZ[0]) );
MKL_LONG is[3], os[3], idist, odist; // input and output strides and distance
...
// on input:  R{k1,k2,m} = AR[is[0] + k1*is[1] + k2*is[2] + m*idist]
status = DftiComputeForward( desc, R, C );
// on output:  Z{k1,k2,m} = AZ[os[0]+k1*os[1]+k2*os[2]+m*odist]
// for k2=0...N2/2:  Z{k1,k2,m} = AZ[os[0]+k1*os[1]+k2*os[2]+m*odist]
// for k2=N2/2+1...N2-1:  Z{k1,k2,m} = conj(AZ[os[0]+(N1-k1)*N1*os[1]+(N2-k2)*N2*os[2]+m*odist])
```

For the backward transform, the input and output parameters and layouts exchange roles: set the strides describing the layout in the backward/forward domain as input/output strides, respectively. For example:

```c
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, bwd_domain_strides );
status = DftiSetValue( desc, DFTI_OUTPUT_STRIDES, fwd_domain_strides );
status = DftiCommitDescriptor( desc );
status = DftiComputeForward( desc, ... );
...
status = DftiSetValue( desc, DFTI_INPUT_STRIDES, bwd_domain_strides );
```
Important
For in-place transforms, ensure the first element of the input data has the same address as the first
element of the output data for each dimension.

See Also
DftiSetValue

DFTI_PACKED_FORMAT
The result of the forward transform of real data is a conjugate-even sequence. Due to the symmetry
property, only a part of the complex-valued sequence is stored in memory. The DFTI_PACKED_FORMAT
configuration parameter defines how the data is packed. Possible values of DFTI_PACKED_FORMAT depend on
the values of the DFTI_CONJUGATE_EVEN_STORAGE configuration parameter:

- DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_COMPLEX.
  The only value of DFTI_PACKED_FORMAT can be DFTI_CCE_FORMAT. You can use this value with
  transforms of any dimension. For a description of the corresponding packed format, see
  DFTI_CONJUGATE_EVEN_STORAGE.
- DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL.
  DFTI_PACKED_FORMAT can be DFTI_CCS_FORMAT, DFTI_PACK_FORMAT, or DFTI_PERM_FORMAT. You can
  use these values with one- and two-dimensional transforms only. The corresponding packed formats are
described below.

NOTE
Although DFTI_CONJUGATE_EVEN_STORAGE=DFTI_COMPLEX_REAL is the default setting for the
DFTI_REAL forward domain, avoid using this storage scheme because it is supported only for one- and
two-dimensional transforms, is incompatible with storage schemes of other FFT libraries, and will be
deprecated in future.
DFTI_CCS_FORMAT for One-dimensional Transforms

The following figure illustrates the storage of a one-dimensional (1D) size-\(N\) conjugate-even sequence in a real array for the CCS, PACK, and PERM packed formats. The CCS format requires an array of size \(N+2\), while the other formats require an array of size \(N\). Zero-based indexing is used.

Storage of a 1D Size-\(N\) Conjugate-even Sequence in a Real Array

<table>
<thead>
<tr>
<th>(n)</th>
<th>(0)</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(\ldots)</th>
<th>(2L-2)</th>
<th>(2L-1)</th>
<th>(2L)</th>
<th>(2L+1)</th>
<th>(2L+2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N=2L)</td>
<td>(R_0)</td>
<td>0</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(\ldots)</td>
<td>(R_{L-1})</td>
<td>(I_{L-1})</td>
<td>(R_L)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>PACK, (N=2L)</td>
<td>(R_0)</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(R_2)</td>
<td>(\ldots)</td>
<td>(I_{L-1})</td>
<td>(R_L)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERM, (N=2L)</td>
<td>(R_0)</td>
<td>(R_L)</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(\ldots)</td>
<td>(R_{L-1})</td>
<td>(I_{L-1})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(N=2L+1)</td>
<td>(R_0)</td>
<td>0</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(\ldots)</td>
<td>(R_{L-1})</td>
<td>(I_{L-1})</td>
<td>(R_L)</td>
<td>(I_L)</td>
<td>not used</td>
</tr>
<tr>
<td>PACK, (N=2L+1)</td>
<td>(R_0)</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(R_2)</td>
<td>(\ldots)</td>
<td>(I_{L-1})</td>
<td>(R_L)</td>
<td>(I_L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PERM, (N=2L+1)</td>
<td>(R_0)</td>
<td>(R_1)</td>
<td>(I_1)</td>
<td>(R_2)</td>
<td>(\ldots)</td>
<td>(I_{L-1})</td>
<td>(R_L)</td>
<td>(I_L)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**NOTE** For storage of a one-dimensional conjugate-even sequence in a real array, CCS is in the same format as CCE.

The real and imaginary parts of the complex-valued conjugate-even sequence \(Z_k\) are located in a real-valued array \(AC\) as illustrated by figure “Storage of a 1D Size-\(N\) Conjugate-even Sequence in a Real Array” and can be used to reconstruct the whole conjugate-even sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*(N+2) )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT );
...
// on input:  R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output:
// for k=0…N/2:  Z{k} = AC[2*k+0] + I*AC[2*k+1]
// for k=N/2+1…N-1: Z{k} = AC[2*(N-k)%N + 0] - I*AC[2*(N-k)%N + 1]
```

DFTI_CCS_FORMAT for Two-dimensional Transforms

**NOTE** The storage of a two-dimensional (2D) sequence in a packed format is deprecated.
The following figure illustrates the storage of a two-dimensional (2D) \( M \)-by-\( N \) conjugate-even sequence in a real array for the CCS packed format. This format requires an array of size \((M+2)\)-by-\((N+2)\). Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result. "n/u" means "not used".

**Storage of a 2D \( M \)-by-\( N \) Conjugate-even Sequence in a Real Array for the CCS Format**

<table>
<thead>
<tr>
<th>( m )</th>
<th>( n=0 )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>2L-2</th>
<th>2L-1</th>
<th>2L</th>
<th>2L+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( R_{0,0} )</td>
<td>0</td>
<td>( R_{0,1} )</td>
<td>( I_{0,1} )</td>
<td>...</td>
<td>( R_{0L-1} )</td>
<td>( I_{0L-1} )</td>
<td>( R_{0L} )</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( R_{1,1} )</td>
<td>( I_{1,1} )</td>
<td>...</td>
<td>( R_{1L-1} )</td>
<td>( I_{1L-1} )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( R_{1,0} )</td>
<td>0</td>
<td>( R_{2,1} )</td>
<td>( I_{2,1} )</td>
<td>...</td>
<td>( R_{2L-1} )</td>
<td>( I_{2L-1} )</td>
<td>( R_{1L} )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( I_{1,0} )</td>
<td>0</td>
<td>( R_{3,1} )</td>
<td>( I_{3,1} )</td>
<td>...</td>
<td>( R_{3L-1} )</td>
<td>( I_{3L-1} )</td>
<td>( I_{1L} )</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2K-2</td>
<td>( R_{K-1,0} )</td>
<td>0</td>
<td>( R_{2K-2,1} )</td>
<td>( I_{2K-2,1} )</td>
<td>...</td>
<td>( R_{2K-2L-1} )</td>
<td>( I_{2K-2L-1} )</td>
<td>( R_{K-1L} )</td>
<td>0</td>
</tr>
<tr>
<td>2K-1</td>
<td>( I_{K-1,0} )</td>
<td>0</td>
<td>( R_{2K-1,1} )</td>
<td>( I_{2K-1,1} )</td>
<td>...</td>
<td>( R_{2K-1L-1} )</td>
<td>( I_{2K-1L-1} )</td>
<td>( I_{K-1L} )</td>
<td>0</td>
</tr>
<tr>
<td>2K</td>
<td>( R_{K,0} )</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>R_{KL}</td>
</tr>
<tr>
<td>2K+1</td>
<td>0</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>0</td>
</tr>
</tbody>
</table>

**CCS, \( M=2K \), \( N=2L \)**

<table>
<thead>
<tr>
<th>( m )</th>
<th>( n=0 )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>2L-2</th>
<th>2L-1</th>
<th>2L</th>
<th>2L+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( R_{3,0} )</td>
<td>0</td>
<td>( R_{0,1} )</td>
<td>( I_{0,1} )</td>
<td>...</td>
<td>( R_{0L-1} )</td>
<td>( I_{0L-1} )</td>
<td>( R_{0L} )</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( R_{1,1} )</td>
<td>( I_{1,1} )</td>
<td>...</td>
<td>( R_{1L-1} )</td>
<td>( I_{1L-1} )</td>
<td>( R_{1L} )</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( R_{1,0} )</td>
<td>0</td>
<td>( R_{2,1} )</td>
<td>( I_{2,1} )</td>
<td>...</td>
<td>( R_{2L-1} )</td>
<td>( I_{2L-1} )</td>
<td>( R_{2L} )</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( I_{1,0} )</td>
<td>0</td>
<td>( R_{3,1} )</td>
<td>( I_{3,1} )</td>
<td>...</td>
<td>( R_{3L-1} )</td>
<td>( I_{3L-1} )</td>
<td>( R_{3L} )</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2K-2</td>
<td>( R_{K-1,0} )</td>
<td>0</td>
<td>( R_{2K-2,1} )</td>
<td>( I_{2K-2,1} )</td>
<td>...</td>
<td>( R_{2K-2L-1} )</td>
<td>( I_{2K-2L-1} )</td>
<td>( R_{2K-2L} )</td>
<td>0</td>
</tr>
<tr>
<td>2K-1</td>
<td>( I_{K-1,0} )</td>
<td>0</td>
<td>( R_{2K-1,1} )</td>
<td>( I_{2K-1,1} )</td>
<td>...</td>
<td>( R_{2K-1L-1} )</td>
<td>( I_{2K-1L-1} )</td>
<td>( I_{2K-1L} )</td>
<td>0</td>
</tr>
<tr>
<td>2K</td>
<td>( R_{K,0} )</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>( n/u )</td>
</tr>
<tr>
<td>2K+1</td>
<td>0</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>( n/u )</td>
</tr>
</tbody>
</table>

**CCS, \( M=2K \), \( N=2L+1 \)**
### Fourier Transform Functions

<table>
<thead>
<tr>
<th>m=0</th>
<th>n=0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>2L-2</th>
<th>2L-1</th>
<th>2L</th>
<th>2L+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{0,0}</td>
<td>0</td>
<td>0</td>
<td>R_{0,1}</td>
<td>I_{0,1}</td>
<td>...</td>
<td>R_{0,L-1}</td>
<td>I_{0,L-1}</td>
<td>R_{0,L}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R_{1,1}</td>
<td>I_{1,1}</td>
<td>...</td>
<td>R_{1,L-1}</td>
<td>I_{1,L-1}</td>
<td>R_{1,L}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>R_{1,0}</td>
<td>0</td>
<td>R_{2,1}</td>
<td>I_{2,1}</td>
<td>...</td>
<td>R_{2,L-1}</td>
<td>I_{2,L-1}</td>
<td>R_{2,L}</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>I_{1,0}</td>
<td>0</td>
<td>R_{3,1}</td>
<td>I_{3,1}</td>
<td>...</td>
<td>R_{3,L-1}</td>
<td>I_{3,L-1}</td>
<td>R_{3,L}</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2K-2</td>
<td>R_{K-1,0}</td>
<td>0</td>
<td>R_{2K-2,1}</td>
<td>I_{2K-2,1}</td>
<td>...</td>
<td>R_{2K-2,L-1}</td>
<td>I_{2K-2,L-1}</td>
<td>R_{2K-2,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K-1</td>
<td>I_{K-1,0}</td>
<td>0</td>
<td>R_{2K-1,1}</td>
<td>I_{2K-1,1}</td>
<td>...</td>
<td>R_{2K-1,L-1}</td>
<td>I_{2K-1,L-1}</td>
<td>R_{2K-1,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K</td>
<td>R_{K-0}</td>
<td>0</td>
<td>R_{2K,1}</td>
<td>I_{2K,1}</td>
<td>...</td>
<td>R_{2K,L-1}</td>
<td>I_{2K,L-1}</td>
<td>R_{2K,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K+1</td>
<td>I_{K-0}</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>I_{K,L}</td>
<td>n/u</td>
</tr>
<tr>
<td>2K+2</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
</tr>
</tbody>
</table>

**CCS, M=2K+1, N=2L**

<table>
<thead>
<tr>
<th>m=0</th>
<th>n=0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>...</th>
<th>2L-2</th>
<th>2L-1</th>
<th>2L</th>
<th>2L+1</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_{0,0}</td>
<td>0</td>
<td>0</td>
<td>R_{0,1}</td>
<td>I_{0,1}</td>
<td>...</td>
<td>R_{0,L-1}</td>
<td>I_{0,L-1}</td>
<td>R_{0,L}</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>R_{1,1}</td>
<td>I_{1,1}</td>
<td>...</td>
<td>R_{1,L-1}</td>
<td>I_{1,L-1}</td>
<td>R_{1,L}</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>R_{1,0}</td>
<td>0</td>
<td>R_{2,1}</td>
<td>I_{2,1}</td>
<td>...</td>
<td>R_{2,L-1}</td>
<td>I_{2,L-1}</td>
<td>R_{2,L}</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>I_{1,0}</td>
<td>0</td>
<td>R_{3,1}</td>
<td>I_{3,1}</td>
<td>...</td>
<td>R_{3,L-1}</td>
<td>I_{3,L-1}</td>
<td>R_{3,L}</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2K-2</td>
<td>R_{K-1,0}</td>
<td>0</td>
<td>R_{2K-2,1}</td>
<td>I_{2K-2,1}</td>
<td>...</td>
<td>R_{2K-2,L-1}</td>
<td>I_{2K-2,L-1}</td>
<td>R_{2K-2,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K-1</td>
<td>I_{K-1,0}</td>
<td>0</td>
<td>R_{2K-1,1}</td>
<td>I_{2K-1,1}</td>
<td>...</td>
<td>R_{2K-1,L-1}</td>
<td>I_{2K-1,L-1}</td>
<td>R_{2K-1,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K</td>
<td>R_{K-0}</td>
<td>0</td>
<td>R_{2K,1}</td>
<td>I_{2K,1}</td>
<td>...</td>
<td>R_{2K,L-1}</td>
<td>I_{2K,L-1}</td>
<td>R_{2K,L}</td>
<td>0</td>
</tr>
<tr>
<td>2K+1</td>
<td>I_{K-0}</td>
<td>0</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>I_{K,L}</td>
<td>n/u</td>
</tr>
<tr>
<td>2K+2</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>...</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
<td>n/u</td>
</tr>
</tbody>
</table>

**CCS, M=2K+1, N=2L+1**
The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in a real-valued array $AC$ as illustrated by figure "Storage of a 2D M-by-N Conjugate-even Sequence in a Real Array for the CCS Format" and can be used to reconstruct the whole sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*(N1+2)*(N2+2) )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_CCS_FORMAT );
...
// on input:  R(k1,k2) = AR[(N2+2)*k1 + k2]
status = DftiComputeForward( desc, AR, AC );  // real-to-complex FFT
// on output: Z(k1,k2) = re + I*im, where
// if (k1==0) {
//   if (k2 <= N2/2) {
//     re = AC[2*k2+0];
//     im = AC[2*k2+1];
//   } else {
//     re = AC[2*(N2-k2)+0];
//     im = -AC[2*(N2-k2)+1];
//   }
// } else if (k2==0){
//   if (k1 <= N1/2) {
//     re = AC[(2*k1+0)*(N2+2)];
//     im = AC[(2*k1+1)*(N2+2)];
//   } else {
//     re = AC[(2*(N1-k1)+0)*(N2+2)];
//     im = -AC[(2*(N1-k1)+1)*(N2+2)];
//   }
// } else if (k2 == N2-k2) {
//   if (k1 <= N1/2) {
//     re = AC[(2*k1+0)*(N2+2) + 2*(N2/2)];
//     im = AC[(2*k1+1)*(N2+2) + 2*(N2/2)];
//   } else{
//     re = AC[(2*(N1-k1)+0)*(N2+2) + 2*(N2/2)];
//     im = -AC[(2*(N1-k1)+1)*(N2+2) + 2*(N2/2)];
//   }
// } else if (k2 <= N2/2) {
//   re = AC[k1*(N2+2)+2*k2+0];
//   im = AC[k1*(N2+2)+2*k2+1];
// } else {
//   re = AC[(N1-k1)*(N2+2)+2*(N2-k2)+0];
//   im = -AC[(N1-k1)*(N2+2)+2*(N2-k2)+1];
// }
```

**DFTI_PACK_FORMAT for One-dimensional Transforms**

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_k$ are located in a real-valued array $AC$ as illustrated by figure "Storage of a 1D Size-$N$ Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT );
...
// on input:  R{k} = AR[k]
status = DftiComputeForward( desc, AR, AC );  // real-to-complex FFT
// on output: Z{k} = re + I*im, where
// if (k == 0) {
```
DFTI_PACK_FORMAT for Two-dimensional Transforms

The following figure illustrates the storage of a 2D $M$-by-$N$ conjugate-even sequence in a real array for the PACK packed format. This format requires an array of size $M$-by-$N$. Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

**Storage of a 2D $M$-by-$N$ Conjugate-even Sequence in a Real Array for the PACK Format**

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_{k1,k2}$ are located in a real-valued array $AC$ as illustrated by figure "Storage of a 2D $M$-by-$N$ Conjugate-even Sequence in a Real Array for the PACK Format" and can be used to reconstruct the whole sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*N1*N2 )
... 
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PACK_FORMAT );
```
... on input: $R_{k1,k2} = AR[N2*k1 + k2]
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
on output: $Z_{k1,k2} = re + I*im$, where
if ($k1==0$) {
    if ($k2 == 0$) {
        re = AC[0];
        im = 0;
    } else if ($k2 == N2-k2$) {
        re = AC[2*k2-1];
        im = 0;
    } else if ($k2 <= N2/2$) {
        re = AC[2*k2-1];
        im = AC[2*k2-0];
    } else {
        re = AC[2*(N2-k2)-1];
        im = -AC[2*(N2-k2)-0];
    }
}
else if ($k2==0$) {
    if ($k1 == N1-k1$) {
        re = AC[(N1-1)*N2];
        im = 0;
    } else if ($k1 <= N1/2$) {
        re = AC[(2*k1-1)*N2];
        im = AC[(2*k1-0)*N2];
    } else {
        re = AC[(2*(N1-k1)-1)*N2];
        im = -AC[(2*(N1-k1)-0)*N2];
    }
}
else if ($k2 == N2-k2$) {
    if ($k1 == N1-k1$) {
        re = AC[N1*N2 - 1];
        im = 0;
    } else if ($k1 <= N1/2$) {
        re = AC[(2*k1 - 1)*N2 + N2-1];
        im = AC[(2*k1 - 0)*N2 + N2-1];
    } else {
        re = AC[(2*(N1-k1) - 1)*N2 + N2-1];
        im = -AC[(2*(N1-k1) - 0)*N2 + N2-1];
    }
}
else if ($k2 <= N2/2$) {
    re = AC[k1*N2+2*k2-1];
    im = AC[k1*N2+2*k2-0];
} else {
    re = AC[(N1-k1)*N2+2*(N2-k2) - 1];
    im = -AC[(N1-k1)*N2+2*(N2-k2) - 0];
}

**DFTI_PERM_FORMAT for One-dimensional Transforms**

The real and imaginary parts of the complex-valued conjugate-even sequence $Z_k$ are located in real-valued array $AC$ as illustrated by figure "Storage of a 1D Size-$N$ Conjugate-even Sequence in a Real Array" and can be used to reconstruct the whole conjugate-even sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N )
float *AC; // malloc( sizeof(float)*N )
```
DFTI_PERM_FORMAT for Two-dimensional Transforms
The following figure illustrates the storage of a 2D \( M \)-by-\( N \) conjugate-even sequence in a real array for the PERM packed format. This format requires an array of size \( M \)-by-\( N \). Row-major layout and zero-based indexing are used. Different colors mark logically separate parts of the result.

Storage of a 2D \( M \)-by-\( N \) Conjugate-Even Sequence in a Real Array for the PERM Format
The real and imaginary parts of the complex-valued conjugate-even sequence \( Z_{k1,k2} \) are located in real-valued array \( AC \) as illustrated by figure "Storage of a 2D-M-by-N Conjugate-even Sequence in a Real Array for the PERM Format" and can be used to reconstruct the whole sequence as follows:

```c
float *AR; // malloc( sizeof(float)*N1*N2 )
float *AC; // malloc( sizeof(float)*N1*N2 )
...
status = DftiSetValue( desc, DFTI_PACKED_FORMAT, DFTI_PERM_FORMAT );
...
// on input: \( R\{k1,k2\} = AR[\{N2\times k1 + k2\}] \)
status = DftiComputeForward( desc, AR, AC ); // real-to-complex FFT
// on output: \( Z\{k1,k2\} = re + I*im, \) where
// if (k1==0) {
  // if (k2 == 0) {
    // re = AC[0];
    // im = 0;
  // } else if (k2 == N2-k2) {
    // re = AC[1];
    // im = 0;
  // } else if (k2 <= N2/2) {
    // re = AC[2*k2+0 - N2%2];
    // im = AC[2*k2+1 - N2%2];
  // } else {
    // re = AC[2*(N2-k2)+0 - N2%2];
    // im = -AC[2*(N2-k2)+1 - N2%2];
  // }
//  } else if (k2==0) {
    // if (k1 == N1-k1) {
      // re = AC[N2];
      // im = 0;
    // } else if (k1 <= N1/2) {
      // re = AC[(2*k1+0 - N1%2)*N2];
      // im = AC[(2*k1+1 - N1%2)*N2];
    // } else {
      // re = AC[(2*(N1-k1)+0 - N1%2)*N2];
      // im = -AC[(2*(N1-k1)+1 - N1%2)*N2];
    // }
//  } else if (k2 == N2-k2) {
    // if (k1 == N1-k1) {
      // re = AC[N2 + 1];
      // im = 0;
    // } else if (k1 <= N1/2) {
      // re = AC[(2*k1+0 - N1%2)*N2 + 1];
      // im = AC[(2*k1+1 - N1%2)*N2 + 1];
    // } else {
      // re = AC[(2*(N1-k1)+0 - N1%2)*N2 + 1];
      // im = -AC[(2*(N1-k1)+1 - N1%2)*N2 + 1];
    // }
//  } else if (k2 <= N2/2) {
    // re = AC[k1*N2+2*k2+0 - N2%2];
    // im = AC[k1*N2+2*k2+1 - N2%2];
  // } else {
    // re = AC[(N1-k1)*N2+2*(N2-k2)+0 - N2%2];
    // im = -AC[(N1-k1)*N2+2*(N2-k2)+1 - N2%2];
  // }
}
```

To better understand packed formats for two-dimensional transforms, see these examples in your Intel® MKL directory:

```
./examples/dftc/source/config_conjugate_even_storage.c
```
**See Also**
DftiSetValue

**DFTI_WORKSPACE**
The computation step for some FFT algorithms requires a scratch space for permutation or other purposes. To manage the use of the auxiliary storage, Intel® MKL enables you to set the configuration parameter `DFTI_WORKSPACE` with the following values:

- **DFTI_ALLOW** (default) Permits the use of the auxiliary storage.
- **DFTI_AVOID** Instructs Intel® MKL to avoid using the auxiliary storage if possible.

**See Also**
DftiSetValue

**DFTI_COMMIT_STATUS**
The `DFTI_COMMIT_STATUS` configuration parameter indicates whether the descriptor is ready for computation. The parameter has two possible values:

- **DFTI_UNCOMMITTED** Default value, set after a successful call of `DftiCreateDescriptor`.
- **DFTI_COMMITTED** The value after a successful call to `DftiCommitDescriptor`.

A computation function called with an uncommitted descriptor returns an error.

You cannot directly set this configuration parameter in a call to `DftiSetValue`, but a change in the configuration of a committed descriptor may change the commit status of the descriptor to `DFTI_UNCOMMITTED`.

**See Also**
DftiCreateDescriptor
DftiCommitDescriptor
DftiSetValue

**DFTI_ORDERING**
Some FFT algorithms apply an explicit permutation stage that is time consuming [4]. The exclusion of this step is similar to applying an FFT to input data whose order is scrambled, or allowing a scrambled order of the FFT results. In applications such as convolution and power spectrum calculation, the order of result or data is unimportant and thus using scrambled data is acceptable if it leads to better performance. The following options are available in Intel® MKL:

- **DFTI_ORDERED**: Forward transform data ordered, backward transform data ordered (default option).
- **DFTI_BACKWARD_SCRAMBLED**: Forward transform data ordered, backward transform data scrambled.

Table "Scrambled Order Transform" tabulates the effect of this configuration setting.

<table>
<thead>
<tr>
<th>Scrambled Order Transform</th>
<th>DftiComputeForward</th>
<th>DftiComputeBackward</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DFTI_ORDERING</strong></td>
<td>Input → Output</td>
<td>Input → Output</td>
</tr>
<tr>
<td><strong>DFTI_ORDERED</strong></td>
<td>ordered → ordered</td>
<td>ordered → ordered</td>
</tr>
<tr>
<td><strong>DFTI_BACKWARD_SCRAMBLED</strong></td>
<td>ordered → scrambled</td>
<td>scrambled → ordered</td>
</tr>
</tbody>
</table>
The word "scrambled" in this table means "permit scrambled order if possible". In some situations permitting out-of-order data gives no performance advantage and an implementation may choose to ignore the suggestion.

See Also
DftiSetValue

FFT Descriptor Manipulation Functions
This category contains the following functions: create a descriptor, commit a descriptor, copy a descriptor, and free a descriptor.

DftiCreateDescriptor
Allocates the descriptor data structure and initializes it with default configuration values.

Syntax
status = DftiCreateDescriptor(&desc_handle, precision, forward_domain, dimension, length);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>precision</td>
<td>enum</td>
<td>Precision of the transform: DFTI_SINGLE or DFTI_DOUBLE.</td>
</tr>
<tr>
<td>forward_domain</td>
<td>enum</td>
<td>Forward domain of the transform: DFTI_COMPLEX or DFTI_REAL.</td>
</tr>
<tr>
<td>dimension</td>
<td>MKL_LONG</td>
<td>Dimension of the transform.</td>
</tr>
<tr>
<td>length</td>
<td>MKL_LONG if dimension = 1. Array of type MKL_LONG otherwise.</td>
<td>Length of the transform for a one-dimensional transform. Lengths of each dimension for a multi-dimensional transform.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>

Description
This function allocates memory for the descriptor data structure and instantiates it with all the default configuration settings for the precision, forward domain, dimension, and length of the desired transform. Because memory is allocated dynamically, the result is actually a pointer to the created descriptor. This function is slightly different from the "initialization" function that can be found in software packages or libraries that implement more traditional algorithms for computing an FFT. This function does not perform
any significant computational work such as computation of twiddle factors. The function
\texttt{DftiCommitDescriptor} does this work after the function \texttt{DftiSetValue} has set values of all necessary
parameters.

The function returns zero when it completes successfully. See \textit{Status Checking Functions} for more
information on the returned status.

\textbf{Prototype}

\begin{verbatim}
/* Note that the preprocessor definition provided below only illustrates
 * that the actual function called may be determined at compile time.
 * You can rely only on the declaration of the function.
 * For precise definition of the preprocessor macro, see the include/mkl_dfti.h
 * file in the Intel MKL directory.
 */
MKL_LONG DftiCreateDescriptor(DFTI_DESCRIPTOR_HANDLE * pHandle,
   enum DFTI_CONFIG_VALUE precision,
   enum DFTI_CONFIG_VALUE domain,
   MKL_LONG dimension, ... /* length/lengths */);

#define DftiCreateDescriptor(desc,prec,domain,dim,sizes) 
   ((prec)==DFTI_SINGLE && (dim)==1) ? 
       some_actual_function_s1d((desc),(domain),(MKL_LONG)(sizes)) : 
       ...
\end{verbatim}

Variable \texttt{length/lengths} is interpreted as a scalar (\texttt{MKL_LONG}) or an array (\texttt{MKL_LONG*}), depending on the
value of parameter \texttt{dimension}. If the value of parameter \texttt{precision} is known at compile time, an optimizing
compiler retains only the call to the respective specific function, thereby reducing the size of the statically
linked application. Avoid direct calls to the specific functions used in the preprocessor macro definition,
because their interface may change in future releases of the library. If the use of the macro is undesirable,
you can safely undefine it after inclusion of the Intel\textsuperscript{®} MKL FFT header file, as follows:

\begin{verbatim}
#include "mkl_dfti.h"
#undef DftiCreateDescriptor
\end{verbatim}

\textbf{See Also}
\texttt{DFTI_PRECISION} configuration parameter
\texttt{DFTI_FORWARD_DOMAIN} configuration parameter
\texttt{DFTI_DIMENSION, DFTI_LENGTHS} configuration parameters

\textbf{Configuration Parameters, summary table}

\textbf{DftiCommitDescriptor}

\textit{Performs all initialization for the actual FFT computation.}

\textbf{Syntax}

\begin{verbatim}
status = DftiCommitDescriptor(desc_handle);
\end{verbatim}

\textbf{Include Files}

\begin{itemize}
  \item mkl.h
\end{itemize}
**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>Updated FFT descriptor.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>

**Description**

This function completes initialization of a previously created descriptor, which is required before the descriptor can be used for FFT computations. Typically, committing the descriptor performs all initialization that is required for the actual FFT computation. The initialization done by the function may involve exploring different factorizations of the input length to find the optimal computation method.

If you call the `DftiSetValue` function to change configuration parameters of a committed descriptor (see Descriptor Configuration Functions), you must re-commit the descriptor before invoking a computation function. Typically, a committal function call is immediately followed by a computation function call (see FFT Computation Functions).

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

**Prototype**

```c
MKL_LONG DftiCommitDescriptor( DFTI_DESCRIPTOR_HANDLE );
```

**DftiFreeDescriptor**

*Frees the memory allocated for a descriptor.*

**Syntax**

```c
status = DftiFreeDescriptor(&desc_handle);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>Memory for the FFT descriptor is released.</td>
</tr>
</tbody>
</table>
### FreeDescriptor

This function frees all memory allocated for a descriptor.

#### Description

Memory allocation/deallocation inside Intel® MKL is managed by Intel® MKL memory management software. So, even after successful completion of `FreeDescriptor`, the memory space may continue being allocated for the application because the memory management software sometimes does not return the memory space to the OS, but considers the space free and can reuse it for future memory allocation. See Example mkl_free_buffers: Usage with FFT Functions in the description of the service function `FreeBuffers` on how to use Intel® MKL memory management software and release memory to the OS.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

#### Prototype

```c
MKL_LONG DftiFreeDescriptor( DFTI_DESCRIPTOR_HANDLE * );
```

#### DftiCopyDescriptor

Makes a copy of an existing descriptor.

#### Syntax

```c
status = DftiCopyDescriptor(desc_handle_original, &desc_handle_copy);
```

#### Include Files

- mkl.h

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle_original</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>The FFT descriptor to copy.</td>
</tr>
<tr>
<td>desc_handle_original</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>The FFT descriptor to copy.</td>
</tr>
</tbody>
</table>

#### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfti_handle_dst</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>The copy of the FFT descriptor.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>
Description

This function makes a copy of an existing descriptor. The resulting descriptor `desc_handle_copy` and the existing descriptor `desc_handle_original` specify the same configuration of the transform, but do not have any memory areas in common ("deep copy").

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Prototype

```c
MKL_LONG DftiCopyDescriptor( DFTI_DESCRIPTOR_HANDLE, DFTI_DESCRIPTOR_HANDLE * );
```

FFT Descriptor Configuration Functions

This category contains the following functions: the value setting function `DftiSetValue` sets one particular configuration parameter to an appropriate value, and the value getting function `DftiGetValue` reads the value of one particular configuration parameter. While all configuration parameters are readable, you cannot set a few of them. Some of these contain fixed information of a particular implementation such as version number, or dynamic information, which is derived by the implementation during execution of one of the functions. See Configuration Settings for details.

**DftiSetValue**

*Sets one particular configuration parameter with the specified configuration value.*

**Syntax**

```c
status = DftiSetValue(desc_handle, config_param, config_val);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
<tr>
<td>config_param</td>
<td>enum</td>
<td>Configuration parameter.</td>
</tr>
<tr>
<td>config_val</td>
<td>Depends on the config</td>
<td>Configuration value.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>Updated FFT descriptor.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>
Description
This function sets one particular configuration parameter with the specified configuration value. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTHS
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_ORDERING

You cannot use the DftiSetValue function to change configuration parameters DFTI_FORWARD_DOMAIN, DFTI_PRECISION, DFTI_DIMENSION, and DFTI_LENGTHS. Use the DftiCreateDescriptor function to set them.

Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in C C++.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Prototype

```c
MKL_LONG DftiSetValue(DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM, ...);
```

See Also
Configuration Settings for more information on configuration parameters.
DftiCreateDescriptor
DftiGetValue

DftiGetValue

Gets the configuration value of one particular configuration parameter.

Syntax

```c
status = DftiGetValue(desc_handle, config_param, &config_val);
```

Include Files

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
<tr>
<td>config_param</td>
<td>enum</td>
<td>Configuration parameter. See Table &quot;Configuration Parameters&quot; for allowable values of config_param.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_val</td>
<td>Depends on the configuration parameter.</td>
<td>Configuration value.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>

Description

This function gets the configuration value of one particular configuration parameter. Each configuration parameter is a named constant, and the configuration value must have the corresponding type, which can be a named constant or a native type. For available configuration parameters and the corresponding configuration values, see:

- DFTI_PRECISION
- DFTI_FORWARD_DOMAIN
- DFTI_DIMENSION, DFTI_LENGTH
- DFTI_PLACEMENT
- DFTI_FORWARD_SCALE, DFTI_BACKWARD_SCALE
- DFTI_THREAD_LIMIT
- DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
- DFTI_NUMBER_OF_TRANSFORMS
- DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
- DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
- DFTI_PACKED_FORMAT
- DFTI_WORKSPACE
- DFTI_COMMIT_STATUS
- DFTI_ORDERING

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

Prototype

```c
MKL_LONG DftiGetValue( DFTI_DESCRIPTOR_HANDLE, DFTI_CONFIG_PARAM , ... );
```

Configuration Settings  for more information on configuration parameters.

DftiSetValue
**FFT Computation Functions**

This category contains the following functions: compute the forward transform and compute the backward transform.

**DftiComputeForward**

*Computes the forward FFT.*

**Syntax**

\[
\text{status} = \text{DftiComputeForward}(\text{desc\_handle}, \text{x\_inout});
\]

\[
\text{status} = \text{DftiComputeForward}(\text{desc\_handle}, \text{x\_in}, \text{y\_out});
\]

\[
\text{status} = \text{DftiComputeForward}(\text{desc\_handle}, \text{xre\_inout}, \text{xim\_inout});
\]

\[
\text{status} = \text{DftiComputeForward}(\text{desc\_handle}, \text{xre\_in}, \text{xim\_in}, \text{yre\_out}, \text{yim\_out});
\]

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
<tr>
<td>x_inout, x_in</td>
<td>Array of type float or double depending on the precision of the transform, specified in the DFTI_PRECISION configuration setting.</td>
<td>Data to be transformed in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xre_inout, xim_inout, xre_in, xim_in</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>Real and imaginary parts of the data to be transformed in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
</tbody>
</table>

The suffix in parameter names corresponds to the value of the configuration parameter DFTI\_PLACEMENT as follows:

- \_inout to DFTI\_INPLACE
- \_in to DFTI\_NOT\_INPLACE

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>y_out</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>The transformed data in case of a real backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xre_inout, xim_inout, yre_out, yim_out</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>Real and imaginary parts of the transformed data in the case of a complex backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>
The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- _inout to DFTI_INPLACE
- _out to DFTI_NOT_INPLACE

**Include Files**

- mkl.h

**Description**

The DftiComputeForward function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, this function computes the forward FFT, that is, the transform with the minus sign in the exponent, \( \delta = -1 \).

The DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration parameters define the layout of the input and output data and must be properly set in a call to the DftiSetValue function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in C C++.

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

**Prototype**

```c
MKL_LONG DftiComputeForward( DFTI_DESCRIPTOR_HANDLE, void*, ... );
```

**See Also**

Configuration Settings
DFTI_FORWARD_DOMAIN
DFTI_PLACEMENT
DFTI_PACKED_FORMAT
DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE
DFTI_DIMENSION, DFTI_LENGTHS
DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE
DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES
DftiComputeBackward
DftiSetValue

**DftiComputeBackward**

*Computes the backward FFT.*

**Syntax**

```c
status = DftiComputeBackward(desc_handle, x_inout);
status = DftiComputeBackward(desc_handle, y_in, x_out);
status = DftiComputeBackward(desc_handle, xre_inout, xim_inout);
status = DftiComputeBackward(desc_handle, yre_in, yim_in, xre_out, xim_out);
```
## Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>desc_handle</td>
<td>DFTI_DESCRIPTOR_HANDLE</td>
<td>FFT descriptor.</td>
</tr>
<tr>
<td>x_inout, y_in</td>
<td>Array of type float or double depending on the precision of the transform, specified in the DFTI_PRECISION configuration setting.</td>
<td>Data to be transformed in case of a real backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xre_inout,</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>Real and imaginary parts of the data to be transformed in the case of a complex backward domain, determined by the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xim_inout,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yre_in, yim_in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- **_inout** to DFTI_INPLACE
- **_in** to DFTI_NOT_INPLACE

## Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_out</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>The transformed data in case of a real forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xre_inout,</td>
<td>Array of type float or double depending on the precision of the transform.</td>
<td>Real and imaginary parts of the transformed data in the case of a complex forward domain, specified in the DFTI_FORWARD_DOMAIN configuration setting.</td>
</tr>
<tr>
<td>xim_inout,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xre_out, xim_out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Function completion status.</td>
</tr>
</tbody>
</table>

The suffix in parameter names corresponds to the value of the configuration parameter DFTI_PLACEMENT as follows:

- **_inout** to DFTI_INPLACE
- **_out** to DFTI_NOT_INPLACE

## Include Files

- mkl.h

## Description

The function accepts the descriptor handle parameter and one or more data parameters. Given a successfully configured and committed descriptor, the DftiComputeBackward function computes the inverse FFT, that is, the transform with the plus sign in the exponent, $\delta = +1$. 
The `DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, and DFTI_CONJUGATE_EVEN_STORAGE configuration` parameters define the layout of the input and output data and must be properly set in a call to the DftiSetValue function.

The FFT descriptor must be properly configured prior to the function call. Function calls needed to configure an FFT descriptor for a particular call to an FFT computation function are summarized in Configuring and Computing an FFT in C C++.

The number and types of the data parameters that the function requires may vary depending on the configuration of the descriptor. This variation is accommodated by variable parameters.

The function returns zero when it completes successfully. See Status Checking Functions for more information on the returned status.

**Prototype**

```c
MKL_LONG DftiComputeBackward( DFTI_DESCRIPTOR_HANDLE, void *, ... );
```

**See Also**

Configuration Settings

- `DFTI_FORWARD_DOMAIN`
- `DFTI_PLACEMENT`
- `DFTI_PACKED_FORMAT`
- `DFTI_COMPLEX_STORAGE, DFTI_REAL_STORAGE, DFTI_CONJUGATE_EVEN_STORAGE`
- `DFTI_DIMENSION, DFTI_LENGTHS`
- `DFTI_INPUT_DISTANCE, DFTI_OUTPUT_DISTANCE`
- `DFTI_INPUT_STRIDES, DFTI_OUTPUT_STRIDES`
- `DftiComputeForward`
- `DftiSetValue`

**Configuring and Computing an FFT in C/C++**

The table below summarizes information on configuring and computing an FFT in C/C++ for all kinds of transforms and possible combinations of input and output domains.

<table>
<thead>
<tr>
<th>FFT to Compute</th>
<th>Input Data</th>
<th>Output Data</th>
<th>Required FFT Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex-to-complex,</td>
<td>Interleaved complex</td>
<td>Interleaved complex</td>
<td>/* Configure a Descriptor */</td>
</tr>
<tr>
<td>in-place, forward or</td>
<td>numbers</td>
<td>numbers</td>
<td>status = DftiCreateDescriptor(&amp;hand, &lt;precision&gt;, DFTI_COMPLEX, &lt;dimension&gt;, &lt;sizes&gt;);</td>
</tr>
<tr>
<td>backward</td>
<td></td>
<td></td>
<td>status = DftiCommitDescriptor(hand);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* Compute an FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeForward(hand, X_inout);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* or backward FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeBackward(hand, X_inout);</td>
</tr>
</tbody>
</table>

<p>| Complex-to-complex,          | Interleaved complex | Interleaved complex  | /* Configure a Descriptor */                                                                |
| out-of-place, forward or     | numbers             | numbers              | status = DftiCreateDescriptor(&amp;hand, &lt;precision&gt;, DFTI_COMPLEX, &lt;dimension&gt;, &lt;sizes&gt;);     |
| backward                     |                     |                      | status = DftiSetValue(hand, DFTI_PLACEMENT,                                              |</p>
<table>
<thead>
<tr>
<th>FFT to Compute</th>
<th>Input Data</th>
<th>Output Data</th>
<th>Required FFT Function Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex-to-complex,</td>
<td>Split-complex</td>
<td>Split-complex</td>
<td>DFTI_NOT_INPLACE);</td>
</tr>
<tr>
<td>in-place, forward or</td>
<td>numbers</td>
<td>numbers</td>
<td>status = DftiCommitDescriptor(hand);</td>
</tr>
<tr>
<td>backward</td>
<td></td>
<td></td>
<td>/* Compute an FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* forward FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeForward(hand, X_in, Y_out);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* or backward FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeBackward(hand, X_in, Y_out);</td>
</tr>
<tr>
<td>Complex-to-complex,</td>
<td>Split-complex</td>
<td>Split-complex</td>
<td>DFTI_NOT_INPLACE);</td>
</tr>
<tr>
<td>out-of-place,</td>
<td>numbers</td>
<td>numbers</td>
<td>status = DftiCommitDescriptor(hand);</td>
</tr>
<tr>
<td>forward or backward</td>
<td></td>
<td></td>
<td>/* Compute an FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* forward FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeForward(hand, Xre_inout, Xim_inout);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>/* or backward FFT */</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiComputeBackward(hand, Xre_inout, Xim_inout);</td>
</tr>
<tr>
<td>Real-to-complex,</td>
<td>Real numbers</td>
<td>Numbers in the</td>
<td>DFTI_REAL, &lt;dimension&gt;, &lt;sizes&gt;);</td>
</tr>
<tr>
<td>in-place, forward</td>
<td></td>
<td>CCE format</td>
<td>status = DftiCreateDescriptor(4hand, &lt;precision&gt;, DFTI_REAL, &lt;dimension&gt;, &lt;sizes&gt;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiSetValue(hand, DFTI_REAL, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiSetValue(hand, DFTI_PACKETED_FORMAT, DFTI_CCE_FORMAT);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>status = DftiSetValue(hand, DFTI_CCE_FORMAT);</td>
</tr>
<tr>
<td>FFT to Compute</td>
<td>Input Data</td>
<td>Output Data</td>
<td>Required FFT Function Calls</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
<td>-------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td><strong>Real-to-complex, out-of-place, forward</strong></td>
<td>Real numbers</td>
<td>Numbers in the CCE format</td>
<td>DFTI_INPUT_STRIDES, &lt;real_strides&gt;); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, &lt;complex_strides&gt;); status = DftiCommitDescriptor(hand); /* Compute an FFT */ status = DftiComputeForward(hand, X_inout);</td>
</tr>
<tr>
<td><strong>Complex-to-real, in-place, backward</strong></td>
<td>Numbers in the CCE format</td>
<td>Real numbers</td>
<td>/* Configure a Descriptor <em>/ status = DftiCreateDescriptor(&amp;hand, &lt;precision&gt;, DFTI_REAL, &lt;dimension&gt;, &lt;sizes&gt;); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_PLACEMENT, DFTI_NOT_INPLACE); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, &lt;real_strides&gt;); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, &lt;complex_strides&gt;); status = DftiCommitDescriptor(hand); /</em> Compute an FFT */ status = DftiComputeBackward(hand, X_inout);</td>
</tr>
<tr>
<td><strong>Complex-to-real, out-of-place, backward</strong></td>
<td>Numbers in the CCE format</td>
<td>Real numbers</td>
<td>/* Configure a Descriptor <em>/ status = DftiCreateDescriptor(&amp;hand, &lt;precision&gt;, DFTI_REAL, &lt;dimension&gt;, &lt;sizes&gt;); status = DftiSetValue(hand, DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX); status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT); status = DftiSetValue(hand, DFTI_INPUT_STRIDES, &lt;complex_strides&gt;); status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, &lt;real_strides&gt;); status = DftiCommitDescriptor(hand); /</em> Compute an FFT */ status = DftiComputeBackward(hand, X_inout);</td>
</tr>
</tbody>
</table>
FFT to Compute   Input Data   Output Data   Required FFT Function Calls

status = DftiSetValue(hand, DFTI_PACKED_FORMAT, DFTI_CCE_FORMAT);
status = DftiSetValue(hand, DFTI_INPUT_STRIDES, <complex_strides>);
status = DftiSetValue(hand, DFTI_OUTPUT_STRIDES, <real_strides>);
status = DftiCommitDescriptor(hand);

/* Compute an FFT */
status = DftiComputeBackward(hand, X_in, Y_out);

You can find C programs that illustrate configuring and computing FFTs in the examples/dftc/subdirectory of your Intel® MKL directory.

Status Checking Functions

All of the descriptor manipulation, FFT computation, and descriptor configuration functions return an integer value denoting the status of the operation. The functions in this category check that status. The first function is a logical function that checks whether the status reflects an error of a predefined class, and the second is an error message function that returns a character string.

DftiErrorClass
Checks whether the status reflects an error of a predefined class.

Syntax

predicate = DftiErrorClass(status, error_class);

Include Files

• mkl.h

Input Parameters

Name    Type       Description
status   MKL_LONG  Completion status of an FFT function.
error_class   MKL_LONG  Predefined error class.

Output Parameters

Name    Type       Description
predicate   MKL_LONG  Result of checking.

Description

The FFT interface in Intel® MKL provides a set of predefined error classes listed in Table "Predefined Error Classes". They are named constants and have the type MKL_LONG.
Predefined Error Classes

<table>
<thead>
<tr>
<th>Named Constants</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_NO_ERROR</td>
<td>No error. The zero status belongs to this class.</td>
</tr>
<tr>
<td>DFTI_MEMORY_ERROR</td>
<td>Usually associated with memory allocation</td>
</tr>
<tr>
<td>DFTI_INVALID_CONFIGURATION</td>
<td>Invalid settings of one or more configuration parameters</td>
</tr>
<tr>
<td>DFTI_INCONSISTENT_CONFIGURATION</td>
<td>Inconsistent configuration or input parameters</td>
</tr>
<tr>
<td>DFTI_NUMBER_OF_THREADS_ERROR</td>
<td>Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function)</td>
</tr>
<tr>
<td>DFTI_MULTITHREADED_ERROR</td>
<td>Usually associated with a value that OMP routines return in case of errors</td>
</tr>
<tr>
<td>DFTI_BAD_DESCRIPTOR</td>
<td>Descriptor is unusable for computation</td>
</tr>
<tr>
<td>DFTI_UNIMPLEMENTED</td>
<td>Unimplemented legitimate settings; implementation dependent</td>
</tr>
<tr>
<td>DFTI_MKL_INTERNAL_ERROR</td>
<td>Internal library error</td>
</tr>
<tr>
<td>DFTI_1D_LENGTH_EXCEEDS_INT32</td>
<td>Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).</td>
</tr>
</tbody>
</table>

The `DftiErrorClass` function returns a non-zero value if the status belongs to the predefined error class. To check whether a function call was successful, call `DftiErrorClass` with a specific error class. However, the zero value of the status belongs to the `DFTI_NO_ERROR` class and thus the zero status indicates successful completion of an operation. See Example "Using Status Checking Functions" for an illustration of correct use of the status checking functions.

**NOTE**
It is incorrect to directly compare a status with a predefined class.

Prototype

```c
MKL_LONG DftiErrorClass( MKL_LONG , MKL_LONG );
```

**DftiErrorMessage**

*Generates an error message.*

**Syntax**

```c
error_message = DftiErrorMessage(status);
```

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>MKL_LONG</td>
<td>Completion status of a function.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>error_message</td>
<td>Array of char</td>
<td>The character string with the error message.</td>
</tr>
</tbody>
</table>

Description

The error message function generates an error message character string. The function returns a pointer to a constant character string, that is, a character array with terminating '\0' character, and you do not need to free this pointer.

Example Using Status Checking Function shows how this function can be used.

Prototype

```c
char *DftiErrorMessage( MKL_LONG );
```

Cluster FFT Functions

This section describes the cluster Fast Fourier Transform (FFT) functions implemented in Intel® MKL.

These functions are available only for Intel® 64 and Intel® Many Integrated Core architectures.

The cluster FFT function library was designed to perform fast Fourier transforms on a cluster, that is, a group of computers interconnected via a network. Each computer (node) in the cluster has its own memory and processor(s). Data interchanges between the nodes are provided by the network.

One or more processes may be running in parallel on each cluster node. To organize communication between different processes, the cluster FFT function library uses the Message Passing Interface (MPI). To avoid dependence on a specific MPI implementation (for example, MPICH, Intel® MPI, and others), the library works with MPI via a message-passing library for linear algebra called BLACS.

Cluster FFT functions of Intel® MKL provide one-dimensional, two-dimensional, and multi-dimensional (up to the order of 7) functions and both Fortran and C interfaces for all transform functions.

To develop applications using the cluster FFT functions, you should have basic skills in MPI programming.

The interfaces for the Intel® MKL cluster FFT functions are similar to the corresponding interfaces for the conventional Intel® MKL FFT functions, described earlier in this chapter. Refer there for details not explained in this section.

Table "Cluster FFT Functions in Intel® MKL" lists cluster FFT functions implemented in Intel® MKL:

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DftiCreateDescriptorDM</td>
<td>Allocates memory for the descriptor data structure and preliminarily initializes it.</td>
</tr>
</tbody>
</table>
### Computing Cluster FFT

The cluster FFT functions described later in this section are provided with Fortran and C interfaces.

Cluster FFT computation is performed by `DftiComputeForwardDM` and `DftiComputeBackwardDM` functions, called in a program using MPI, which will be referred to as MPI program. After an MPI program starts, a number of processes are created. MPI identifies each process by its rank. The processes are independent of one another and communicate via MPI. A function called in an MPI program is invoked in all the processes. Each process manipulates data according to its rank. Input or output data for a cluster FFT transform is a sequence of real or complex values. A cluster FFT computation function operates on the local part of the input data, that is, some part of the data to be operated in a particular process, as well as generates local part of the output data. While each process performs its part of computations, running in parallel and communicating through MPI, the processes perform the entire FFT computation. FFT computations using the Intel® MKL cluster FFT functions are typically effected by a number of steps listed below:

1. Initiate MPI by calling `MPI_Init` (the function must be called prior to calling any FFT function and any MPI function).
2. Allocate memory for the descriptor and create it by calling `DftiCreateDescriptorDM`.
3. Specify one of several values of configuration parameters by one or more calls to `DftiSetValueDM`.
4. Obtain values of configuration parameters needed to create local data arrays; the values are retrieved by calling `DftiGetValueDM`.
5. Initialize the descriptor for the FFT computation by calling `DftiCommitDescriptorDM`.
6. Create arrays for local parts of input and output data and fill the local part of input data with values. (For more information, see **Distributing Data among Processes**.)
7. Compute the transform by calling `DftiComputeForwardDM` or `DftiComputeBackwardDM`.
8. Gather local output data into the global array using MPI functions. (This step is optional because you may need to immediately employ the data differently.)
10. Finalize communication through MPI by calling `MPI_Finalize` (the function must be called after the last call to a cluster FFT function and the last call to an MPI function).

Several code examples in the "Examples for Cluster FFT Functions" section in the Code Examples appendix illustrate cluster FFT computations.

### Distributing Data among Processes

The Intel® MKL cluster FFT functions store all input and output multi-dimensional arrays (matrices) in one-dimensional arrays (vectors). The arrays are stored in the row-major order. For example, a two-dimensional matrix \( A \) of size \((m, n)\) is stored in a vector \( B \) of size \( m*n \) so that

\[
B[i*n+j] = A[i][j] \quad (i=0, \ldots, m-1, \ j=0, \ldots, n-1)
\]
Order of FFT dimensions is the same as the order of array dimensions in the programming language. For example, a 3-dimensional FFT with Lengths=(m,n,l) can be computed over an array \( A_r[m][n][l] \).

All MPI processes involved in cluster FFT computation operate their own portions of data. These local arrays make up the virtual global array that the fast Fourier transform is applied to. It is your responsibility to properly allocate local arrays (if needed), fill them with initial data and gather resulting data into an actual global array or process the resulting data differently. To be able do this, see sections below on how the virtual global array is composed of the local ones.

**Multi-dimensional transforms**

If the dimension of transform is greater than one, the cluster FFT function library splits data in the dimension whose index changes most slowly, so that the parts contain all elements with several consecutive values of this index. It is the first dimension in C. If the global array is two-dimensional, it gives each process several consecutive rows. Local arrays are placed in memory allocated for the virtual global array consecutively, in the order determined by process ranks. For example, in case of two processes, during the computation of a three-dimensional transform whose matrix has size (11,15,12), the processes may store local arrays of sizes (6,15,12) and (5,15,12), respectively.

If \( p \) is the number of MPI processes and the matrix of a transform to be computed has size \((m,n,l)\), each MPI process works with local data array of size \((m_q, n, l)\), where \( \sum m_q = m, \) \( q=0, \ldots, p-1 \). Local input arrays must contain appropriate parts of the actual global input array, and then local output arrays will contain appropriate parts of the actual global output array. You can figure out which particular rows of the global array the local array must contain from the following configuration parameters of the cluster FFT interface: CDFT_LOCAL_NX, CDFT_LOCAL_START_X, and CDFT_LOCAL_SIZE. To retrieve values of the parameters, use the DftiGetValueDM function:

- **CDFT_LOCAL_NX** specifies how many rows of the global array the current process receives.
- **CDFT_LOCAL_START_X** specifies which row of the global input or output array corresponds to the first row of the local input or output array. If \( A \) is a global array and \( L \) is the appropriate local array, then

\[
L[i][j][k] = A[i+cdft_local_start_x][j][k], \text{ where } i = 0, \ldots, m_q-1, \ j = 0, \ldots, n-1, \ k = 0, \ldots, l-1.
\]

Example "2D Out-of-place Cluster FFT Computation" in the Code Examples appendix shows how the data is distributed among processes for a two-dimensional cluster FFT computation.

**One-dimensional transforms**

In this case, input and output data are distributed among processes differently and even the numbers of elements stored in a particular process before and after the transform may be different. Each local array stores a segment of consecutive elements of the appropriate global array. Such segment is determined by the number of elements and a shift with respect to the first array element. So, to specify segments of the global input and output arrays that a particular process receives, four configuration parameters are needed: CDFT_LOCAL_NX, CDFT_LOCAL_START_X, CDFT_LOCAL_OUT_NX, and CDFT_LOCAL_OUT_START_X. Use the DftiGetValueDM function to retrieve their values. The meaning of the four configuration parameters depends upon the type of the transform, as shown in Table "Data Distribution Configuration Parameters for 1D Transforms":

<table>
<thead>
<tr>
<th>Meaning of the Parameter</th>
<th>Forward Transform</th>
<th>Backward Transform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of elements in input array</td>
<td>CDFT_LOCAL_NX</td>
<td>CDFT_LOCAL_OUT_NX</td>
</tr>
<tr>
<td>Elements shift in input array</td>
<td>CDFT_LOCAL_START_X</td>
<td>CDFT_LOCAL_OUT_START_X</td>
</tr>
</tbody>
</table>
Meaning of the Parameter | Forward Transform | Backward Transform |
--- | --- | --- |
Number of elements in output array | CDFT_LOCAL_OUT_NX | CDFT_LOCAL_NX |
Elements shift in output array | CDFT_LOCAL_OUT_START_X | CDFT_LOCAL_START_X |

Memory size for local data

The memory size needed for local arrays cannot be just calculated from CDFT_LOCAL_NX (CDFT_LOCAL_OUT_NX), because the cluster FFT functions sometimes require allocating a little bit more memory for local data than just the size of the appropriate sub-array. The configuration parameter CDFT_LOCAL_SIZE specifies the size of the local input and output array in data elements. Each local input and output arrays must have size not less than CDFT_LOCAL_SIZE * size_of_element. Note that in the current implementation of the cluster FFT interface, data elements can be real or complex values, each complex value consisting of the real and imaginary parts. If you employ a user-defined workspace for in-place transforms (for more information, refer to Table "Settable configuration Parameters"), it must have the same size as the local arrays. Example "1D In-place Cluster FFT Computations" in the Code Examples appendix illustrates how the cluster FFT functions distribute data among processes in case of a one-dimensional FFT computation performed with a user-defined workspace.

Available Auxiliary Functions

If a global input array is located on one MPI process and you want to obtain its local parts or you want to gather the global output array on one MPI process, you can use functions MKL_CDFT_ScatterData and MKL_CDFT_GatherData to distribute or gather data among processes, respectively. These functions are defined in a file that is delivered with Intel® MKL and located in the following subdirectory of the Intel® MKL installation directory: examples/cdftc/source/cdft_example_support.c.

Restriction on Lengths of Transforms

The algorithm that the Intel® MKL cluster FFT functions use to distribute data among processes imposes a restriction on lengths of transforms with respect to the number of MPI processes used for the FFT computation:

- For a multi-dimensional transform, lengths of the first two dimensions must be not less than the number of MPI processes.
- Length of a one-dimensional transform must be the product of two integers each of which is not less than the number of MPI processes.

Non-compliance with the restriction causes an error CDFT_SPREAD_ERROR (refer to Error Codes for details). To achieve the compliance, you can change the transform lengths and/or the number of MPI processes, which is specified at start of an MPI program. MPI-2 enables changing the number of processes during execution of an MPI program.

Cluster FFT Interface

To use the cluster FFT functions, you need to access the header file mkl_cdtf.h through "include".

The C interface provides a structure type DFTI_DESCRIPTOR_DM_HANDLE and a number of functions, some of which accept a different number of input arguments.

To provide communication between parallel processes through MPI, the following include statement must be present in your code:

- C/C++:
  ```c
  #include "mpi.h"
  ```
There are three main categories of the cluster FFT functions in Intel® MKL:

1. **Descriptor Manipulation.** There are three functions in this category. The DftiCreateDescriptorDM function creates an FFT descriptor whose storage is allocated dynamically. The DftiCommitDescriptorDM function "commits" the descriptor to all its settings. The DftiFreeDescriptorDM function frees up the memory allocated for the descriptor.

2. **FFT Computation.** There are two functions in this category. The DftiComputeForwardDM function performs the forward FFT computation, and the DftiComputeBackwardDM function performs the backward FFT computation.

3. **Descriptor Configuration.** There are two functions in this category. The DftiSetValueDM function sets one specific configuration value to one of the many configuration parameters. The DftiGetValueDM function gets the current value of any of these configuration parameters, all of which are readable. These parameters, though many, are handled one at a time.

**Cluster FFT Descriptor Manipulation Functions**

There are three functions in this category: create a descriptor, commit a descriptor, and free a descriptor.

**DftiCreateDescriptorDM**

*Allocates memory for the descriptor data structure and preliminarily initializes it.*

**Syntax**

\[
\text{status} = \text{DftiCreateDescriptorDM}(\text{comm}, \&\text{handle}, \text{v1}, \text{v2}, \text{dim}, \text{size});
\]

\[
\text{status} = \text{DftiCreateDescriptorDM}(\text{comm}, \&\text{handle}, \text{v1}, \text{v2}, \text{dim}, \text{sizes});
\]

**Include Files**

- mkl_cdft.h

**Input Parameters**

- **comm**: MPI communicator, e.g. MPI_COMM_WORLD.
- **v1**: Precision of the transform.
- **v2**: Type of the forward domain. Must be DFTI_COMPLEX for complex-to-complex transforms or DFTI_REAL for real-to-complex transforms.
- **dim**: Dimension of the transform.
- **size**: Length of the transform in a one-dimensional case.
- **sizes**: Lengths of the transform in a multi-dimensional case.

**Output Parameters**

- **handle**: Pointer to the descriptor handle of transform. If the function completes successfully, the pointer to the created handle is stored in the variable.

**Description**

This function allocates memory in a particular MPI process for the descriptor data structure and instantiates it with default configuration settings with respect to the precision, domain, dimension, and length of the desired transform. The domain is understood to be the domain of the forward transform. The result is a pointer to the created descriptor. This function is slightly different from the "initialization" function.
DftiCommitDescriptorDM in a more traditional software packages or libraries used for computing the FFT. This function does not perform any significant computation work, such as twiddle factors computation, because the default configuration settings can still be changed using the function DftiSetValueDM.

The value of the parameter \( v1 \) is specified through named constants DFTI_SINGLE and DFTI_DOUBLE. It corresponds to precision of input data, output data, and computation. A setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The parameter \( dim \) is a simple positive integer indicating the dimension of the transform.

For one-dimensional transforms, length is a single integer value of the parameter \( size \) having type MKL_LONG; for multi-dimensional transforms, length is supplied with the parameter \( sizes \), which is an array of integers having type MKL_LONG.

**Return Values**

The function returns DFTI_NO_ERROR when completes successfully. In this case, the pointer to the created descriptor handle is stored in \( handle \). If the function fails, it returns a value of another error class constant

**Prototype**

```c
MKL_LONG DftiCreateDescriptorDM(MPI_Comm,DFTI_DESCRIPTOR_DM_HANDLE*,
                                    enum DFTI_CONFIG_VALUE,enum DFTI_CONFIG_VALUE,MKL_LONG,...);
```

**Include Files**

- mkl_cdft.h

**Input Parameters**

- **handle**

**Description**

The cluster FFT interface requires a function that completes initialization of a previously created descriptor before the descriptor can be used for FFT computations in a particular MPI process. The DftiCommitDescriptorDM function performs all initialization that facilitates the actual FFT computation. For the current implementation, it may involve exploring many different factorizations of the input length to search for highly efficient computation method.

Any changes of configuration parameters of a committed descriptor via the set value function (see Descriptor Configuration Functions) requires a re-committal of the descriptor before a computation function can be invoked. Typically, this committal function is called right before a computation function call (see FFT Computation Functions).

**Return Values**

The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).
Prototype

MKL_LONG DftiCommitDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE handle);

DftiFreeDescriptorDM
Frees memory allocated for a descriptor.

Syntax
status = DftiFreeDescriptorDM(&handle);

Include Files
• mkl_cdft.h

Input Parameters
handle
The descriptor handle. Must be valid, that is, created in a call to DftiCreateDescriptorDM.

Output Parameters
handle
The descriptor handle. Memory allocated for the handle is released on output.

Description
This function frees up all memory allocated for a descriptor in a particular MPI process. Call the DftiFreeDescriptorDM function to delete the descriptor handle. Upon successful completion of DftiFreeDescriptorDM the descriptor handle is no longer valid.

Return Values
The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Prototype

MKL_LONG DftiFreeDescriptorDM(DFTI_DESCRIPTOR_DM_HANDLE *handle);

Cluster FFT Computation Functions
There are two functions in this category: compute the forward transform and compute the backward transform.

DftiComputeForwardDM
Computes the forward FFT.

Syntax
status = DftiComputeForwardDM(handle, in_X, out_X);
status = DftiComputeForwardDM(handle, in_out_X);
Include Files
• mkl_cdft.h

Input Parameters

handle
The descriptor handle.

in_X, in_out_X
Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X
Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.

Description

The DftiComputeForwardDM function computes the forward FFT. Forward FFT is the transform using the factor \( e^{-i \frac{2\pi}{n}} \).

Before you call the function, the valid descriptor, created by DftiCreateDescriptorDM, must be configured and committed using the DftiCommitDescriptorDM function.

The computation is carried out by calling the DftiComputeForward function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of DftiComputeForward.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the Distributing Data Among Processes section for details.

Refer to the Configuration Settings section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter DFTI_PRECISION determines the precision of input data, output data, and transform: a setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.

The configuration parameter DFTI_PLACEMENT informs the function whether the computation should be in-place. If the value of this parameter is DFTI_INPLACE (default), you must call the function with two parameters, otherwise you must supply three parameters. If DFTI_PLACEMENT = DFTI_INPLACE and three parameters are supplied, then the third parameter is ignored.

Caution
Even in case of an out-of-place transform, local array of input data in_X may be changed. To save data, make its copy before calling DftiComputeForwardDM.

In case of an in-place transform, DftiComputeForwardDM dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

NOTE
You can specify your own workspace of the same size through the configuration parameter CDFT_WORKSPACE to avoid redundant memory allocation.
Return Values
The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Prototype

MKL_LONG DftiComputeForwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);

DftiComputeBackwardDM
Computes the backward FFT.

Syntax

status = DftiComputeBackwardDM(handle, in_X, out_X);
status = DftiComputeBackwardDM(handle, in_out_X);

Include Files
• mkl_cdft.h

Input Parameters

handle
The descriptor handle.
in_X, in_out_X
Local part of input data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate and initialize the array.

Output Parameters

out_X, in_out_X
Local part of output data. Array of complex values. Refer to the Distributing Data among Processes section on how to allocate the array.

Description
The DftiComputeBackwardDM function computes the backward FFT. Backward FFT is the transform using the factor $e^{i2\pi/n}$.

Before you call the function, the valid descriptor, created by DftiCreateDescriptorDM, must be configured and committed using the DftiCommitDescriptorDM function.

The computation is carried out by calling the DftiComputeBackward function. So, the functions have very much in common, and details not explicitly mentioned below can be found in the description of DftiComputeBackward.

Local part of input data, as well as local part of the output data, is an appropriate sequence of complex values (each complex value consists of two real numbers: real part and imaginary part) that a particular process stores. See the Distributing Data among Processes section for details.

Refer to the Configuration Settings section for the list of configuration parameters that the descriptor passes to the function.

The configuration parameter DFTI_PRECISION determines the precision of input data, output data, and transform: a setting of DFTI_SINGLE indicates single-precision floating-point data type and a setting of DFTI_DOUBLE indicates double-precision floating-point data type.
The configuration parameter DFTI_PLACEMENT informs the function whether the computation should be in-place. If the value of this parameter is DFTI_INPLACE (default), you must call the function with two parameters, otherwise you must supply three parameters. If DFTI_PLACEMENT = DFTI_INPLACE and three parameters are supplied, then the third parameter is ignored.

**Caution**
Even in case of an out-of-place transform, local array of input data \( in_X \) may be changed. To save data, make its copy before calling DftiComputeBackwardDM.

In case of an in-place transform, DftiComputeBackwardDM dynamically allocates and deallocates a work buffer of the same size as the local input/output array requires.

**NOTE**
You can specify your own workspace of the same size through the configuration parameter CDFT_WORKSPACE to avoid redundant memory allocation.

**Return Values**
The function returns DFTI_NO_ERROR when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

**Prototype**

```c
MKL_LONG DftiComputeBackwardDM(DFTI_DESCRIPTOR_DM_HANDLE handle, void *in_X,...);
```

**Cluster FFT Descriptor Configuration Functions**
There are two functions in this category: the value setting function DftiSetValueDM sets one particular configuration parameter to an appropriate value, the value getting function DftiGetValueDM reads the value of one particular configuration parameter.

Some configuration parameters used by cluster FFT functions originate from the conventional FFT interface (see Configuration Settings subsection in the "FFT Functions" section for details).

Other configuration parameters are specific to the cluster FFT. Integer values of these parameters have type MKL_LONG. The exact type of the configuration parameters being floating-point scalars is float or double. The configuration parameters whose values are named constants have the enum type. They are defined in the mkl_cdft.h header file.

Names of the configuration parameters specific to the cluster FFT interface have prefix CDFT.

**DftiSetValueDM**
Sets one particular configuration parameter with the specified configuration value.

**Syntax**

```c
status = DftiSetValueDM (handle, param, value);
```

**Include Files**

- mkl_cdft.h
Input Parameters

**handle**
The descriptor handle. Must be valid, that is, created in a call to `DftiCreateDescriptorDM`.

**param**
Name of a parameter to be set up in the descriptor handle. See Table "Settable Configuration Parameters" for the list of available parameters.

**value**
Value of the parameter.

Description

This function sets one particular configuration parameter with the specified configuration value. The configuration parameter is one of the named constants listed in the table below, and the configuration value must have the corresponding type. See Configuration Settings for details of the meaning of each setting and for possible values of the parameters whose values are named constants.

Settable Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Data Type</th>
<th>Description</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_FORWARD_SCALE</td>
<td>Floating-point scalar</td>
<td>Scale factor of forward transform.</td>
<td>1.0</td>
</tr>
<tr>
<td>DFTI_BACKWARD_SCALE</td>
<td>Floating-point scalar</td>
<td>Scale factor of backward transform.</td>
<td>1.0</td>
</tr>
<tr>
<td>DFTI_PLACEMENT</td>
<td>Named constant</td>
<td>Placement of the computation result.</td>
<td>DFTI_INPLACE</td>
</tr>
<tr>
<td>DFTI_ORDERING</td>
<td>Named constant</td>
<td>Scrambling of data order.</td>
<td>DFTI_ORDERED</td>
</tr>
<tr>
<td>CDFT_WORKSPACE</td>
<td>Array of an appropriate type</td>
<td>Auxiliary buffer, a user-defined workspace. Enables saving memory during in-place computations.</td>
<td>NULL (allocate workspace dynamically).</td>
</tr>
<tr>
<td>DFTI_PACKED_FORMAT</td>
<td>Named constant</td>
<td>Packed format, real data.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DFTI_PERM_FORMAT — default and the only available value for one-dimensional transforms</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DFTI_CCE_FORMAT — default and the only available value for multi-dimensional transforms</td>
<td></td>
</tr>
<tr>
<td>DFTI_TRANSPOSE</td>
<td>Named constant</td>
<td>This parameter determines how the output data is located for multi-dimensional transforms. If the parameter value is DFTI_NONE, the data is located in a usual manner described in this document. If the value is DFTI_ALLOW, the</td>
<td>DFTI_NONE</td>
</tr>
</tbody>
</table>
Return Values
The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

Prototype

```c
MKL_LONG DftiSetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param, ...);
```

### DftiGetValueDM

*Gets the value of one particular configuration parameter.*

**Syntax**

```c
status = DftiGetValueDM(handle, param, &value);
```

**Include Files**
- `mkl_cdft.h`

**Input Parameters**

- `handle` The descriptor handle. Must be valid, that is, created in a call to `DftiCreateDescriptorDM`.
- `param` Name of a parameter to be retrieved from the descriptor. See Table "Retrievable Configuration Parameters" for the list of available parameters.

**Output Parameters**

- `value` Value of the parameter.

**Description**

This function gets the configuration value of one particular configuration parameter. The configuration parameter is one of the named constants listed in the table below, and the configuration value is the corresponding appropriate type, which can be a named constant or a native type. Possible values of the named constants can be found in Table "Configuration Parameters" and relevant subsections of the Configuration Settings section.

#### Retrievable Configuration Parameters

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_PRECISION</td>
<td>Named constant</td>
<td>Precision of computation, input data and output data.</td>
</tr>
<tr>
<td>DFTI_DIMENSION</td>
<td>Integer scalar</td>
<td>Dimension of the transform</td>
</tr>
<tr>
<td>Parameter Name</td>
<td>Data Type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>----------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DFTI_LENGTHS</td>
<td>Array of integer values</td>
<td>Array of lengths of the transform. Number of lengths corresponds to the dimension of the transform.</td>
</tr>
<tr>
<td>DFTI_FORWARD_SCALE</td>
<td>Floating-point scalar</td>
<td>Scale factor of forward transform.</td>
</tr>
<tr>
<td>DFTI_BACKWARD_SCALE</td>
<td>Floating-point scalar</td>
<td>Scale factor of backward transform.</td>
</tr>
<tr>
<td>DFTI_PLACEMENT</td>
<td>Named constant</td>
<td>Placement of the computation result.</td>
</tr>
<tr>
<td>DFTI_COMMIT_STATUS</td>
<td>Named constant</td>
<td>Shows whether descriptor has been committed.</td>
</tr>
<tr>
<td>DFTI_FORWARD_DOMAIN</td>
<td>Named constant</td>
<td>Forward domain of transforms, has the value of DFTI_COMPLEX or DFTI_REAL.</td>
</tr>
<tr>
<td>DFTI_ORDERING</td>
<td>Named constant</td>
<td>Scrambling of data order.</td>
</tr>
<tr>
<td>CDFT_MPI_COMM</td>
<td>Type of MPI communicator</td>
<td>MPI communicator used for transforms.</td>
</tr>
<tr>
<td>CDFT_LOCAL_SIZE</td>
<td>Integer scalar</td>
<td>Necessary size of input, output, and buffer arrays in data elements.</td>
</tr>
<tr>
<td>CDFT_LOCAL_X_START</td>
<td>Integer scalar</td>
<td>Row/element number of the global array that corresponds to the first row/element of the local array. For more information, see Distributing Data among Processes.</td>
</tr>
<tr>
<td>CDFT_LOCAL_NX</td>
<td>Integer scalar</td>
<td>The number of rows/elements of the global array stored in the local array. For more information, see Distributing Data among Processes.</td>
</tr>
<tr>
<td>CDFT_LOCAL_OUT_X_START</td>
<td>Integer scalar</td>
<td>Element number of the appropriate global array that corresponds to the first element of the input or output local array in a 1D case. For details, see Distributing Data among Processes.</td>
</tr>
<tr>
<td>CDFT_LOCAL_OUT_NX</td>
<td>Integer scalar</td>
<td>The number of elements of the appropriate global array that are stored in the input or output local array in a 1D case. For details, see Distributing Data among Processes.</td>
</tr>
</tbody>
</table>

**Return Values**

The function returns `DFTI_NO_ERROR` when completes successfully. If the function fails, it returns a value of another error class constant (for the list of constants, refer to the Error Codes section).

**Prototype**

```c
MKL_LONG DftiGetValueDM(DFTI_DESCRIPTOR_DM_HANDLE handle, int param,...);
```
**Error Codes**

All the cluster FFT functions return an integer value denoting the status of the operation. These values are identified by named constants. Each function returns `DFTI_NO_ERROR` if no errors were encountered during execution. Otherwise, a function generates an error code. In addition to FFT error codes, the cluster FFT interface has its own ones. Named constants specific to the cluster FFT interface have prefix "CDFT" in names. Table "Error Codes that Cluster FFT Functions Return" lists error codes that the cluster FFT functions may return.

<table>
<thead>
<tr>
<th>Named Constants</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTI_NO_ERROR</td>
<td>No error.</td>
</tr>
<tr>
<td>DFTI_MEMORY_ERROR</td>
<td>Usually associated with memory allocation.</td>
</tr>
<tr>
<td>DFTI_INVALID_CONFIGURATION</td>
<td>Invalid settings of one or more configuration parameters.</td>
</tr>
<tr>
<td>DFTI_INCONSISTENT_CONFIGURATION</td>
<td>Inconsistent configuration or input parameters.</td>
</tr>
<tr>
<td>DFTI_NUMBER_OF_THREADS_ERROR</td>
<td>Number of OMP threads in the computation function is not equal to the number of OMP threads in the initialization stage (commit function).</td>
</tr>
<tr>
<td>DFTI_MULTITHREADED_ERROR</td>
<td>Usually associated with a value that OMP routines return in case of errors.</td>
</tr>
<tr>
<td>DFTI_BAD_DESCRIPTOR</td>
<td>Descriptor is unusable for computation.</td>
</tr>
<tr>
<td>DFTI_UNIMPLEMENTED</td>
<td>Unimplemented legitimate settings; implementation dependent.</td>
</tr>
<tr>
<td>DFTI_MKL_INTERNAL_ERROR</td>
<td>Internal library error.</td>
</tr>
<tr>
<td>DFTI_1D_LENGTH_EXCEEDS_INT32</td>
<td>Length of one of dimensions exceeds $2^{32} - 1$ (4 bytes).</td>
</tr>
<tr>
<td>CDFT_SPREAD_ERROR</td>
<td>Data cannot be distributed (For more information, see Distributing Data among Processes.)</td>
</tr>
<tr>
<td>CDFT_MPI_ERROR</td>
<td>MPI error. Occurs when calling MPI.</td>
</tr>
</tbody>
</table>
This chapter describes the Intel® Math Kernel Library implementation of the PBLAS (Parallel Basic Linear Algebra Subprograms) routines from the ScalAPACK package for distributed-memory architecture. PBLAS is intended for using in vector-vector, matrix-vector, and matrix-matrix operations to simplify the parallelization of linear codes. The design of PBLAS is as consistent as possible with that of the BLAS. The routine descriptions are arranged in several sections according to the PBLAS level of operation:

- PBLAS Level 1 Routines (distributed vector-vector operations)
- PBLAS Level 2 Routines (distributed matrix-vector operations)
- PBLAS Level 3 Routines (distributed matrix-matrix operations)

Each section presents the routine and function group descriptions in alphabetical order by the routine group name; for example, the `p?asum` group, the `p?axpy` group. The question mark in the group name corresponds to a character indicating the data type (s, d, c, and z or their combination); see Routine Naming Conventions.

**NOTE**
PBLAS routines are provided only with Intel® MKL versions for Linux* and Windows* OSs.

Generally, PBLAS runs on a network of computers using MPI as a message-passing layer and a set of prebuilt communication subprograms (BLACS), as well as a set of PBLAS optimized for the target architecture. The Intel® MKL version of PBLAS is optimized for Intel® processors. For the detailed system and environment requirements see Intel® MKL Release Notes and Intel® MKL Developer Guide.

For full reference on PBLAS routines and related information, see http://www.netlib.org/scalapack/html/pblas_qref.html.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**PBLAS Routines Overview**

The model of the computing environment for PBLAS is represented as a one-dimensional array of processes or also a two-dimensional process grid. To use PBLAS, all global matrices or vectors must be distributed on this array or grid prior to calling the PBLAS routines.

PBLAS uses the two-dimensional block-cyclic data distribution as a layout for dense matrix computations. This distribution provides good work balance between available processors, as well as gives the opportunity to use PBLAS Level 3 routines for optimal local computations. Information about the data distribution that is required to establish the mapping between each global array and its corresponding process and memory location is contained in the so called *array descriptor* associated with each global array. Table "Content of the array descriptor for dense matrices" gives an example of an array descriptor structure.
### Content of Array Descriptor for Dense Matrices

<table>
<thead>
<tr>
<th>Array Element #</th>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dtype</td>
<td>Descriptor type (=1 for dense matrices)</td>
</tr>
<tr>
<td>2</td>
<td>ctxt</td>
<td>BLACS context handle for the process grid</td>
</tr>
<tr>
<td>3</td>
<td>m</td>
<td>Number of rows in the global array</td>
</tr>
<tr>
<td>4</td>
<td>n</td>
<td>Number of columns in the global array</td>
</tr>
<tr>
<td>5</td>
<td>mb</td>
<td>Row blocking factor</td>
</tr>
<tr>
<td>6</td>
<td>nb</td>
<td>Column blocking factor</td>
</tr>
<tr>
<td>7</td>
<td>rsrc</td>
<td>Process row over which the first row of the global array is distributed</td>
</tr>
<tr>
<td>8</td>
<td>csrc</td>
<td>Process column over which the first column of the global array is distributed</td>
</tr>
<tr>
<td>9</td>
<td>lld</td>
<td>Leading dimension of the local array</td>
</tr>
</tbody>
</table>

The number of rows and columns of a global dense matrix that a particular process in a grid receives after data distributing is denoted by $\text{LOCr}(\cdot)$ and $\text{LOCc}(\cdot)$, respectively. To compute these numbers, you can use the ScaLAPACK tool routine `numroc`.

After the block-cyclic distribution of global data is done, you may choose to perform an operation on a submatrix of the global matrix $A$, which is contained in the global subarray $\text{sub}(A)$, defined by the following 6 values (for dense matrices):

- $m$: The number of rows of $\text{sub}(A)$
- $n$: The number of columns of $\text{sub}(A)$
- $a$: A pointer to the local array containing the entire global array $A$
- $ia$: The row index of $\text{sub}(A)$ in the global array
- $ja$: The column index of $\text{sub}(A)$ in the global array
- $\text{desca}$: The array descriptor for the global array $A$

Intel® MKL provides the PBLAS routines with interface similar to the interface used in the Netlib PBLAS (see http://www.netlib.org/scalapack/html/pblas_qref.html).

## PBLAS Routine Naming Conventions

The naming convention for PBLAS routines is similar to that used for BLAS routines (see Routine Naming Conventions in Chapter 2). A general rule is that each routine name in PBLAS, which has a BLAS equivalent, is simply the BLAS name prefixed by initial letter $p$ that stands for "parallel".

The Intel® MKL PBLAS routine names have the following structure:

```
p <character> <name> <mod> ( )
```

The `<character>` field indicates the Fortran data type:

- s: real, single precision
- c: complex, single precision
- d: real, double precision
- z: complex, double precision
- i: integer

Some routines and functions can have combined character codes, such as sc or dz.

For example, the function `pscacum` uses a complex input array and returns a real value.

The `<name>` field, in PBLAS level 1, indicates the operation type. For example, the PBLAS level 1 routines `p?dot`, `p?swap`, `p?copy` compute a vector dot product, vector swap, and a copy vector, respectively.
In PBLAS level 2 and 3, `<name>` reflects the matrix argument type:

- `ge` general matrix
- `sy` symmetric matrix
- `he` Hermitian matrix
- `tr` triangular matrix

In PBLAS level 3, the `<name>=tran` indicates the transposition of the matrix.

The `<mod>` field, if present, provides additional details of the operation. The PBLAS level 1 names can have the following characters in the `<mod>` field:

- `c` conjugated vector
- `u` unconjugated vector

The PBLAS level 2 names can have the following additional characters in the `<mod>` field:

- `mv` matrix-vector product
- `sv` solving a system of linear equations with matrix-vector operations
- `r` rank-1 update of a matrix
- `r2` rank-2 update of a matrix.

The PBLAS level 3 names can have the following additional characters in the `<mod>` field:

- `mm` matrix-matrix product
- `sm` solving a system of linear equations with matrix-matrix operations
- `rk` rank-`k` update of a matrix
- `r2k` rank-`2k` update of a matrix.

The examples below show how to interpret PBLAS routine names:

- `pddot` `<p> <d> <dot>`: double-precision real distributed vector-vector dot product
- `pcdotc` `<p> <c> <dot> <c>`: complex distributed vector-vector dot product, conjugated
- `pcasum` `<p> <sc> <asum>`: sum of magnitudes of distributed vector elements, single precision real output and single precision complex input
- `pcdotu` `<p> <c> <dot> <u>`: distributed vector-vector dot product, unconjugated, complex
- `psgemv` `<p> <s> <ge> <mv>`: distributed matrix-vector product, general matrix, single precision
- `pztrmm` `<p> <z> <tr> <mm>`: distributed matrix-matrix product, triangular matrix, double-precision complex.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
Optimization Notice

optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

PBLAS Level 1 Routines

PBLAS Level 1 includes routines and functions that perform distributed vector-vector operations. Table "PBLAS Level 1 Routine Groups and Their Data Types" lists the PBLAS Level 1 routine groups and the data types associated with them.

PBLAS Level 1 Routine Groups and Their Data Types

<table>
<thead>
<tr>
<th>Routine or Function Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p?amax</td>
<td>s, d, c, z</td>
<td>Calculates an index of the distributed vector element with maximum absolute value</td>
</tr>
<tr>
<td>p?asum</td>
<td>s, d, sc, dz</td>
<td>Calculates sum of magnitudes of a distributed vector</td>
</tr>
<tr>
<td>p?axpy</td>
<td>s, d, c, z</td>
<td>Calculates distributed vector-scalar product</td>
</tr>
<tr>
<td>p?copy</td>
<td>s, d, c, z</td>
<td>Copies a distributed vector</td>
</tr>
<tr>
<td>p?dot</td>
<td>s, d</td>
<td>Calculates a dot product of two distributed real vectors</td>
</tr>
<tr>
<td>p?dotc</td>
<td>c, z</td>
<td>Calculates a dot product of two distributed complex vectors, one of them is conjugated</td>
</tr>
<tr>
<td>p?dotu</td>
<td>c, z</td>
<td>Calculates a dot product of two distributed complex vectors</td>
</tr>
<tr>
<td>p?nrm2</td>
<td>s, d, sc, dz</td>
<td>Calculates the 2-norm (Euclidean norm) of a distributed vector</td>
</tr>
<tr>
<td>p?scal</td>
<td>s, d, c, z, cs, zd</td>
<td>Calculates a product of a distributed vector by a scalar</td>
</tr>
<tr>
<td>p?swap</td>
<td>s, d, c, zd</td>
<td>Swaps two distributed vectors</td>
</tr>
</tbody>
</table>

p?amax

Computes the global index of the element of a distributed vector with maximum absolute value.

Syntax

void psamax (const MKL_INT *n, float *amax, MKL_INT *indx, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pdamax (const MKL_INT *n, double *amax, MKL_INT *indx, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pcamax (const MKL_INT *n, MKL_Complex8 *amax, MKL_INT *indx, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pzamax (const MKL_INT *n, MKL_Complex16 *amax, MKL_INT *indx, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

**Include Files**
- mkl_pblas.h

**Description**
The functions pamax compute global index of the maximum element in absolute value of a distributed vector sub(x), where sub(x) denotes X(ix, jx:jx+n-1) if incx=m_x, and X(ix: ix+n-1, jx) if incx=1.

**Input Parameters**

\( n \)  
(global) The length of distributed vector sub(x), \( n \geq 0 \).

\( x \)  
(local)  
Array, size \((jx-1)*m_x + ix+(n-1)*abs(incx))\).  
This array contains the entries of the distributed vector sub(x).

\( ix, jx \)  
(global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix sub(X), respectively.

\( descx \)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix X.

\( incx \)  
(global) Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and \( m_x \). incx must not be zero.

**Output Parameters**

\( amax \)  
(global).  
Maximum absolute value (magnitude) of elements of the distributed vector only in its scope.

\( indx \)  
(global) The global index of the maximum element in absolute value of the distributed vector sub(x) only in its scope.

**p?asum**
*Computes the sum of magnitudes of elements of a distributed vector.*

**Syntax**

void psasum (const MKL_INT *n, float *asum, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pdasum (const MKL_INT *n, double *asum, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pscasum (const MKL_INT *n, float *asum, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pdzasum (const MKL_INT *n, double *asum, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

Include Files

- mkl_pblas.h

Description

The functions p?asum compute the sum of the magnitudes of elements of a distributed vector \( \text{sub}(x) \), where \( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx}=m_x \), and \( X(ix: ix+n-1, jx) \) if \( \text{incx}=1 \).

Input Parameters

\( n \)  
(general) The length of distributed vector \( \text{sub}(x) \), \( n \geq 0 \).

\( x \)  
(local)  
Array, size \( (jx-1)*m_x + ix+(n-1)*\text{abs}(\text{incx}) \).  
This array contains the entries of the distributed vector \( \text{sub}(x) \).

\( ix, jx \)  
(general) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(X) \), respectively.

\( \text{descx} \)  
(general and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

\( \text{incx} \)  
(general) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

Output Parameters

\( \text{asum} \)  
(local) and \( \text{pscasum} \).  
Contains the sum of magnitudes of elements of the distributed vector only in its scope.

\textit{p?axpy}

\textit{Computes a distributed vector-scalar product and adds the result to a distributed vector.}

Syntax

void psaxpy (const MKL_INT *n, const float *a, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pdaxpy (const MKL_INT *n, const double *a, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pcaxpy (const MKL_INT *n, const MKL_Complex8 *a, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
void pzaxpy (const MKL_INT *n, const MKL_Complex16 *a, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

**Include Files**
- mkl_pblas.h

**Description**
The `p?axpy` routines perform the following operation with distributed vectors:

\[
\text{sub}(y) := \text{sub}(y) + a \cdot \text{sub}(x)
\]

where:
- \(a\) is a scalar;
- \(\text{sub}(x)\) and \(\text{sub}(y)\) are \(n\)-element distributed vectors.
- \(\text{sub}(x)\) denotes \(X(ix, jx: jx+n-1)\) if \(incx=m_x\), and \(X(ix: ix+n-1, jx)\) if \(incx=1\);
- \(\text{sub}(y)\) denotes \(Y(iy, jy: jy+n-1)\) if \(incy=m_y\), and \(Y(iy: iy+n-1, jy)\) if \(incy=1\).

**Input Parameters**

- \(n\) (global) The length of distributed vectors, \(n \geq 0\).
- \(a\) (local)
  Specifies the scalar \(a\).
- \(x\) (local)
  Array, size \((jx-1) \cdot m_x + ix+(n-1) \cdot \text{abs}(incx))\).
  This array contains the entries of the distributed vector \(\text{sub}(x)\).
- \(ix, jx\) (global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix \(\text{sub}(X)\), respectively.
- \(\text{descx}\) (global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).
- \(incx\) (global) Specifies the increment for the elements of \(\text{sub}(x)\). Only two values are supported, namely 1 and \(m_x\). \(incx\) must not be zero.
- \(y\) (local)
  Array, size \((jy-1) \cdot m_y + iy+(n-1) \cdot \text{abs}(incy))\).
  This array contains the entries of the distributed vector \(\text{sub}(y)\).
- \(iy, jy\) (global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \(\text{sub}(Y)\), respectively.
- \(\text{descy}\) (global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).
- \(incy\) (global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two values are supported, namely 1 and \(m_y\). \(incy\) must not be zero.
Output Parameters

\[ y \quad \text{Overwritten by } sub(y) := sub(y) + a \cdot sub(x). \]

**p?copy**

*Copies one distributed vector to another vector.*

**Syntax**

```c
void picopy (const MKL_INT *n , const MKL_INT *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , MKL_INT *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
void pscopy (const MKL_INT *n , const float *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , float *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
void pdcopy (const MKL_INT *n , const double *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , double *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
void pccopy (const MKL_INT *n , const MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , MKL_Complex8 *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
void pzcopy (const MKL_INT *n , const MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx ,
const MKL_INT *descx , const MKL_INT *incx , MKL_Complex16 *y , const MKL_INT *iy ,
const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

**Include Files**

- mkl_pblas.h

**Description**

The **p?copy** routines perform a copy operation with distributed vectors defined as

\[ sub(x) = sub(y), \]

where \( sub(x) \) and \( sub(y) \) are \( n \)-element distributed vectors.

\( sub(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( incx = m_x \) and \( X(ix: ix+n-1, jx) \) if \( incx = 1 \);

\( sub(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( incy = m_y \) and \( Y(iy: iy+n-1, jy) \) if \( incy = 1 \).

**Input Parameters**

- \( n \)
  
  (global) The length of distributed vectors, \( n \geq 0 \).

- \( x \)
  
  (local)

  Array, size \((jx-1)*m_x + ix+(n-1)*abs(incx))\).

  This array contains the entries of the distributed vector \( sub(x) \).

- \( ix, jx \)
  
  (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( sub(X) \), respectively.

- \( descx \)
  
  (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).
The \texttt{?dot} functions compute the dot product \texttt{dot} of two distributed real vectors defined as

\[
dot = \text{sub}(x)' \ast \text{sub}(y)
\]

where \text{sub}(x) and \text{sub}(y) are \(n\)-element distributed vectors.

\text{sub}(x) denotes \(X(ix, jx:jx+n-1)\) if \(incx=m_x\), and \(X(ix: ix+n-1, jx)\) if \(incx=1\);

\text{sub}(y) denotes \(Y(iy, jy:jy+n-1)\) if \(incy=m_y\), and \(Y(iy: iy+n-1, jy)\) if \(incy=1\).

**Input Parameters**

\(n\) (global) The length of distributed vectors, \(n \geq 0\).

\(x\) (local) Array, size \((jx-1)*m_x + ix+(n-1)*\text{abs}(incx))\).

\(y\) (local) Array, size \((jy-1)*m_y + iy+(n-1)*\text{abs}(incy))\).

This array contains the entries of the distributed vector \text{sub}(y).

\(iy, jy\) (global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \text{sub}(Y), respectively.

\(descy\) (global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).

\(incy\) (global) Specifies the increment for the elements of \text{sub}(y). Only two values are supported, namely 1 and \(m_y\). \(incy\) must not be zero.

**Output Parameters**

\(y\) Overwritten with the distributed vector \text{sub}(x).
This array contains the entries of the distributed vector \( \text{sub}(x) \).

\( ix, jx \) (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(X) \), respectively.

\( \text{descx} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

\( \text{incy} \) (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incy} \) must not be zero.

\( y \) (local)

Array, size \((jy-1)*m_y + iy + (n-1)*\text{abs(incy)})\).

This array contains the entries of the distributed vector \( \text{sub}(y) \).

\( iy, jy \) (global) The row and column indices in the distributed matrix \( Y \) indicating the first row and the first column of the submatrix \( \text{sub}(Y) \), respectively.

\( \text{descy} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( Y \).

\( \text{incy} \) (global) Specifies the increment for the elements of \( \text{sub}(y) \). Only two values are supported, namely 1 and \( m_y \). \( \text{incy} \) must not be zero.

**Output Parameters**

\( \text{dot} \) (local)

Dot product of \( \text{sub}(x) \) and \( \text{sub}(y) \) only in their scope.

**p?dotc**

*Computes the dot product of two distributed complex vectors, one of them is conjugated.*

**Syntax**

```c
void pcdotc (const MKL_INT *n, MKL_Complex8 *dotc, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy);
void pzdotc (const MKL_INT *n, MKL_Complex16 *dotc, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy);
```

**Include Files**

- `mkl_pblas.h`

**Description**

The \( p?\text{dotc} \) functions compute the dot product \( \text{dotc} \) of two distributed vectors, with one vector conjugated:

\[
\text{dotc} = \text{conjg(sub(x)')}*\text{sub(y)}
\]

where \( \text{sub}(x) \) and \( \text{sub}(y) \) are \( n \)-element distributed vectors.
sub($x$) denotes $X(ix, jx:jx+n-1)$ if $incx = m_x$, and $X(ix: ix+n-1, jx)$ if $incx = 1$;
sub($y$) denotes $Y(iy, jy:jy+n-1)$ if $incy = m_y$, and $Y(iy: iy+n-1, jy)$ if $incy = 1$.

**Input Parameters**

$n$  
(global) The length of distributed vectors, $n \geq 0$.

$x$  
(local)
Array, size $(jx-1) \times m_x + ix + \lfloor (n-1) \times abs(incx) \rfloor$.
This array contains the entries of the distributed vector sub($x$).

$ix, jx$  
(global) The row and column indices in the distributed matrix $X$ indicating the first row and the first column of the submatrix sub($X$), respectively.

descx  
(global and local) array of dimension 9. The array descriptor of the distributed matrix $X$.

$incx$  
(global) Specifies the increment for the elements of sub($x$). Only two values are supported, namely 1 and $m_x$. $incx$ must not be zero.

$y$  
(local)
Array, size $(jy-1) \times m_y + iy + \lfloor (n-1) \times abs(incy) \rfloor$.
This array contains the entries of the distributed vector sub($y$).

$iy, jy$  
(global) The row and column indices in the distributed matrix $Y$ indicating the first row and the first column of the submatrix sub($Y$), respectively.

descy  
(global and local) array of dimension 9. The array descriptor of the distributed matrix $Y$.

$incy$  
(global) Specifies the increment for the elements of sub($y$). Only two values are supported, namely 1 and $m_y$. $incy$ must not be zero.

**Output Parameters**

dotc  
(local)
Dot product of sub($x$) and sub($y$) only in their scope.

$p\text{dotu}$

*Computes the dot product of two distributed complex vectors.*

**Syntax**

```c
void pcdotu (const MKL_INT *n, MKL_Complex8 *dotu, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
void pzdotu (const MKL_INT *n, MKL_Complex16 *dotu, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
```
Include Files

• mkl_pblas.h

Description

The p?dotu functions compute the dot product \( \text{dotu} \) of two distributed vectors defined as

\[
\text{dotu} = \text{sub}(x)' \ast \text{sub}(y)
\]

where \( \text{sub}(x) \) and \( \text{sub}(y) \) are \( n \)-element distributed vectors.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx} = m_x \), and \( X(ix: ix+n-1, jx) \) if \( \text{incx} = 1 \);

\( \text{sub}(y) \) denotes \( Y(iy, jy: jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy: iy+n-1, jy) \) if \( \text{incy} = 1 \).

Input Parameters

\( n \) (global) The length of distributed vectors, \( n \geq 0 \).

\( x \) (local)

Array, size \( (jx-1) \ast m_x + ix+(n-1) \ast \text{abs(incx)} \).

This array contains the entries of the distributed vector \( \text{sub}(x) \).

\( ix, jx \) (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(X) \), respectively.

\( \text{descx} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

\( \text{incx} \) (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

\( y \) (local)

Array, size \( (jy-1) \ast m_y + iy+(n-1) \ast \text{abs(incy)} \).

This array contains the entries of the distributed vector \( \text{sub}(y) \).

\( iy, jy \) (global) The row and column indices in the distributed matrix \( Y \) indicating the first row and the first column of the submatrix \( \text{sub}(Y) \), respectively.

\( \text{descy} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( Y \).

\( \text{incy} \) (global) Specifies the increment for the elements of \( \text{sub}(y) \). Only two values are supported, namely 1 and \( m_y \). \( \text{incy} \) must not be zero.

Output Parameters

\( \text{dotu} \) (local)

Dot product of \( \text{sub}(x) \) and \( \text{sub}(y) \) only in their scope.

\[ \text{p?nrm2} \]

*Computes the Euclidean norm of a distributed vector.*
Syntax

```c
void psnrm2 (const MKL_INT *n, float *norm2, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pdnrm2 (const MKL_INT *n, double *norm2, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pscnrm2 (const MKL_INT *n, float *norm2, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pdznrm2 (const MKL_INT *n, double *norm2, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
```

Include Files

- mkl_pblas.h

Description

The p?nrm2 functions compute the Euclidean norm of a distributed vector `sub(x)`, where `sub(x)` is an n-element distributed vector.

`sub(x)` denotes `X(ix, jx:jx+n-1)` if `incx=m_x`, and `X(ix: ix+n-1, jx)` if `incx=1`.

Input Parameters

- `n` (global) The length of distributed vector `sub(x)`, n\(\geq 0\).
- `x` (local)
  Array, size \((jx-1)*m_x + ix+(n-1)*\text{abs}(incx))\).
  This array contains the entries of the distributed vector `sub(x)`.
- `ix, jx` (global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix `sub(X)`, respectively.
- `descx` (global and local) array of dimension 9. The array descriptor of the distributed matrix X.
- `incx` (global) Specifies the increment for the elements of `sub(x)`. Only two values are supported, namely 1 and \(m_x\). `incx` must not be zero.

Output Parameters

- `norm2` (local) and `pscnrm2`.
  Contains the Euclidean norm of a distributed vector only in its scope.

p?scal

*Computes a product of a distributed vector by a scalar.*

Syntax

```c
void psscal (const MKL_INT *n, const float *a, float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
```
void pdscal (const MKL_INT *n, const double *a, double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pcscal (const MKL_INT *n, const MKL_Complex8 *a, MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pzscal (const MKL_INT *n, const MKL_Complex16 *a, MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pcsscal (const MKL_INT *n, const float *a, MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pzdscal (const MKL_INT *n, const double *a, MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

Include Files

• mkl_pblas.h

Description

The \texttt{p\textasciitilde scal} routines multiplies a \(n\)-element distributed vector \(\text{sub}(\text{x})\) by the scalar \(a\):

\[ \text{sub}(\text{x}) = a \times \text{sub}(\text{x}), \]

where \(\text{sub}(\text{x})\) denotes \(X(\text{ix}, \text{jx}\text{;}jx+n-1)\) if \(\text{incx}=m_x\), and \(X(\text{ix};\text{ix}+n-1, \text{jx})\) if \(\text{incx}=1\).

Input Parameters

\begin{itemize}
  \item \(n\) (global) The length of distributed vector \(\text{sub}(\text{x})\), \(n \geq 0\).
  \item \(a\) (global) and \texttt{pcsscal}
    Specifies the scalar \(a\).
  \item \(x\) (local)
    Array, size \((jx-1)\times m_x + \text{ix}+(n-1)\times \text{abs(incx)})\).
    This array contains the entries of the distributed vector \(\text{sub}(\text{x})\).
  \item \(\text{ix}, jx\) (global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix \(\text{sub}(X)\), respectively.
  \item \(\text{descx}\) (global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).
  \item \(\text{incx}\) (global) Specifies the increment for the elements of \(\text{sub}(\text{x})\). Only two values are supported, namely 1 and \(m_x\). \(\text{incx}\) must not be zero.
\end{itemize}

Output Parameters

\begin{itemize}
  \item \(x\) Overwritten by the updated distributed vector \(\text{sub}(\text{x})\)
\end{itemize}

\texttt{p\textasciitilde swap}

\textit{Swaps two distributed vectors.}
Syntax

void psswap (const MKL_INT *n, float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_INT *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pdswap (const MKL_INT *n, double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pcswap (const MKL_INT *n, MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pzswap (const MKL_INT *n, MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

Include Files

• mkl_pblas.h

Description

Given two distributed vectors sub(x) and sub(y), the p?swap routines return vectors sub(y) and sub(x) swapped, each replacing the other.

Here sub(x) denotes \(X(\text{ix}, \text{jx}:\text{ix}+\text{n}-1)\) if \(\text{incx}=\text{m}_x\), and \(X(\text{ix}+\text{n}-1, \text{jx})\) if \(\text{incx}=1\);

sub(y) denotes \(Y(\text{iy}, \text{jy}:\text{iy}+\text{n}-1)\) if \(\text{incy}=\text{m}_y\), and \(Y(\text{iy}+\text{n}-1, \text{jy})\) if \(\text{incy}=1\).

Input Parameters

\(n\)  
(global) The length of distributed vectors, \(n\geq 0\).

\(x\)  
(local)  
Array, size \((\text{jx}-1)\text{m}_x + \text{ix}+(\text{n}-1)\text{abs(incx)})\).

This array contains the entries of the distributed vector sub(x).

\(\text{ix, jx}\)  
(global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix sub(X), respectively.

\(\text{descx}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).

\(\text{incy}\)  
(global) Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and \(\text{m}_x\). incx must not be zero.

\(y\)  
(local)  
Array, size \((\text{jy}-1)\text{m}_y + \text{iy}+(\text{n}-1)\text{abs(incy)})\).

This array contains the entries of the distributed vector sub(y).

\(\text{iy, jy}\)  
(global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix sub(Y), respectively.

\(\text{descy}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).
**incy**

Specifies the increment for the elements of \( \text{sub}(y) \). Only two values are supported, namely 1 and \( m_y \). \( incy \) must not be zero.

**Output Parameters**

- \( x \): Overwritten by distributed vector \( \text{sub}(y) \).
- \( y \): Overwritten by distributed vector \( \text{sub}(x) \).

## PBLAS Level 2 Routines

This section describes PBLAS Level 2 routines, which perform distributed matrix-vector operations. Table "PBLAS Level 2 Routine Groups and Their Data Types" lists the PBLAS Level 2 routine groups and the data types associated with them.

### PBLAS Level 2 Routine Groups and Their Data Types

<table>
<thead>
<tr>
<th>Routine Groups</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p?gemv</td>
<td>s, d, c, z</td>
<td>Matrix-vector product using a distributed general matrix</td>
</tr>
<tr>
<td>p?agemv</td>
<td>s, d, c, z</td>
<td>Matrix-vector product using absolute values for a distributed general matrix</td>
</tr>
<tr>
<td>p?ger</td>
<td>s, d</td>
<td>Rank-1 update of a distributed general matrix</td>
</tr>
<tr>
<td>p?gerc</td>
<td>c, z</td>
<td>Rank-1 update (conjugated) of a distributed general matrix</td>
</tr>
<tr>
<td>p?geru</td>
<td>c, z</td>
<td>Rank-1 update (unconjugated) of a distributed general matrix</td>
</tr>
<tr>
<td>p?hemv</td>
<td>c, z</td>
<td>Matrix-vector product using a distributed Hermitian matrix</td>
</tr>
<tr>
<td>p?ahemv</td>
<td>c, z</td>
<td>Matrix-vector product using absolute values for a distributed Hermitian matrix</td>
</tr>
<tr>
<td>p?her</td>
<td>c, z</td>
<td>Rank-1 update of a distributed Hermitian matrix</td>
</tr>
<tr>
<td>p?her2</td>
<td>c, z</td>
<td>Rank-2 update of a distributed Hermitian matrix</td>
</tr>
<tr>
<td>p?syr</td>
<td>s, d</td>
<td>Rank-1 update of a distributed symmetric matrix</td>
</tr>
<tr>
<td>p?syr2</td>
<td>s, d</td>
<td>Rank-2 update of a distributed symmetric matrix</td>
</tr>
<tr>
<td>p?trmv</td>
<td>s, d, c, z</td>
<td>Distributed matrix-vector product using a triangular matrix</td>
</tr>
<tr>
<td>p?atrmv</td>
<td>s, d, c, z</td>
<td>Distributed matrix-vector product using absolute values for a triangular matrix</td>
</tr>
<tr>
<td>p?trsv</td>
<td>s, d, c, z</td>
<td>Solves a system of linear equations whose coefficients are in a distributed triangular matrix</td>
</tr>
</tbody>
</table>
**p?gemv**

*Computes a distributed matrix-vector product using a general matrix.*

**Syntax**

```c
void psgemv(const char *trans, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *x, const MKL_INT *ix, const MKL_INT *ja, const MKL_INT *descx, const MKL_INT *incx, const float *beta, float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pdgemv(const char *trans, const MKL_INT *m, const MKL_INT *n, const float *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *x, const MKL_INT *ix, const MKL_INT *ja, const MKL_INT *descx, const MKL_INT *incx, const double *beta, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pcgemv(const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *ja, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex8 *beta, MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pzgemv(const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *ja, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex16 *beta, MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
```

**Include Files**

- mkl_pblas.h

**Description**

The p?gemv routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(y) := \alpha \cdot \text{sub}(A) \cdot \text{sub}(x) + \beta \cdot \text{sub}(y),
\]

or

\[
\text{sub}(y) := \alpha \cdot \text{sub}(A)^* \cdot \text{sub}(x) + \beta \cdot \text{sub}(y),
\]

or

\[
\text{sub}(y) := \alpha \cdot \text{conjg} \left( \text{sub}(A)^* \right) \cdot \text{sub}(x) + \beta \cdot \text{sub}(y),
\]

where

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(A) \) is a \( m \)-by-\( n \) submatrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) and \( \text{sub}(y) \) are subvectors.

When \( \text{trans} = \text{'N'} \) or \( \text{'n'} \), \( \text{sub}(x) \) denotes \( X(ix, ja:ja+n-1) \) if \( incx = m \_x \), and \( X(ix, ix+n-1, ja) \) if \( incx = 1 \), \( \text{sub}(y) \) denotes \( Y(iy, ja:ja+m-1) \) if \( incy = m \_y \), and \( Y(iy, iy+m-1, ja) \) if \( incy = 1 \).
When \( \text{trans} = 'T' \) or \( 't' \), or \( 'C' \), or \( 'c' \), \( \text{sub}(x) \) denotes \( X(ix, jx:jx+m-1) \) if \( \text{incx} = m_x \), and \( X(ix:ix+m-1, jx) \) if \( \text{incx} = 1 \); \( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy:iy+m-1, jy) \) if \( \text{incy} = 1 \).

**Input Parameters**

- **trans**  
  (global) Specifies the operation:
  
  - if \( \text{trans} = 'N' \) or \( 'n' \), then \( \text{sub}(y) := \alpha \text{sub}(A)' \text{sub}(x) + \beta \text{sub}(y) \);
  
  - if \( \text{trans} = 'T' \) or \( 't' \), then \( \text{sub}(y) := \alpha \text{sub}(A)' \text{sub}(x) + \beta \text{sub}(y) \);
  
  - if \( \text{trans} = 'C' \) or \( 'c' \), then \( \text{sub}(y) := \alpha \text{conjg}(\text{sub}(A))' \text{sub}(x) + \beta \text{sub}(y) \).

- **m**  
  (global) Specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **n**  
  (global) Specifies the number of columns of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha**  
  (global) Specifies the scalar \( \alpha \).

- **a**  
  (local)  
  Array, size \((\text{lld}_a, \text{LOCq}(ja+n-1))\). Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(A) \).

- **ia, ja**  
  (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

- **desca**  
  (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

- **x**  
  (local)  
  Array, size \((\text{ix}+n-1) \times m_x + (\text{jx}+n-1) \times \text{abs(incx)} \) when \( \text{trans} = 'N' \) or \( 'n' \), and \((\text{ix}-1) \times m_x + (\text{jx}-1) \times \text{abs(incx)} \) otherwise.  
  
  This array contains the entries of the distributed vector \( \text{sub}(x) \).

- **ix, jx**  
  (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

- **descx**  
  (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

- **incx**  
  (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

- **beta**  
  (global)  
  Specifies the scalar \( \beta \). When \( \beta \) is set to zero, then \( \text{sub}(y) \) need not be set on input.

- **y**  
  (local)
Array, size \((jy-1)*m_y + iy+(m-1)*\text{abs}(incy)\) when \(\text{trans} = 'N'\) or 'n', and \((jy-1)*m_y + iy+(n-1)*\text{abs}(incy)\) otherwise.

This array contains the entries of the distributed vector \(\text{sub}(y)\).

\(iy, jy\) (global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \(\text{sub}(y)\), respectively.

\(\text{descy}\) (global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).

\(incy\) (global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two values are supported, namely 1 and \(m_y\). \(incy\) must not be zero.

**Output Parameters**

\(y\) Overwritten by the updated distributed vector \(\text{sub}(y)\).

**p?agemv**

*Computes a distributed matrix-vector product using absolute values for a general matrix.*

**Syntax**

```c
void psagemv (const char *trans, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
void pdagemv (const char *trans, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const double *beta, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
void pcagemv (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex8 *beta, const MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
void pzagemv (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex16 *beta, const MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);
```

**Include Files**

- mkl_pblas.h
**Description**

The `pagemv` routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(y) := |\alpha| \cdot |\text{sub}(A)'| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)|,
\]

or

\[
\text{sub}(y) := |\alpha| \cdot |\text{sub}(A)'| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)|,
\]

or

\[
\text{sub}(y) := |\alpha| \cdot |\text{conjg}(\text{sub}(A))'| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)|,
\]

where

\( \alpha \) and \( \beta \) are scalars,

\( \text{sub}(A) \) is a \( m \)-by-\( n \) submatrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \),

\( \text{sub}(x) \) and \( \text{sub}(y) \) are subvectors.

**Input Parameters**

- **trans** (global) Specifies the operation:
  - if `trans` = 'N' or 'n', then \( \text{sub}(y) := |\alpha| \cdot |\text{sub}(A)| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)| \)
  - if `trans` = 'T' or 't', then \( \text{sub}(y) := |\alpha| \cdot |\text{sub}(A)'| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)| \)
  - if `trans` = 'C' or 'c', then \( \text{sub}(y) := |\alpha| \cdot |\text{conjg}(\text{sub}(A))'| \cdot |\text{sub}(x)| + |\beta| \cdot |\text{sub}(y)| \).

- **m** (global) Specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( m \geq 0 \).

- **n** (global) Specifies the number of columns of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha** (global)
  - Specifies the scalar \( \alpha \).

- **a** (local)
Array, size ($lld_a$, LOCq($ja+n-1$)). Before entry this array must contain the local pieces of the distributed matrix $sub(A)$.

$ia, ja$ (global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

$x$ (local)

Array, size $(jx-1)*m_x + ix+(n-1)*abs(incx))$ when trans = 'N' or 'n', and $(jx-1)*m_x + ix+(m-1)*abs(incx))$ otherwise.

This array contains the entries of the distributed vector $sub(x)$.

$ix, jx$ (global) The row and column indices in the distributed matrix $X$ indicating the first row and the first column of the submatrix $sub(x)$, respectively.

descx (global and local) array of dimension 9. The array descriptor of the distributed matrix $X$.

$incx$ (global) Specifies the increment for the elements of $sub(x)$. Only two values are supported, namely 1 and $m_x$. incx must not be zero.

$beta$ (global)

Specifies the scalar $beta$. When $beta$ is set to zero, then $sub(y)$ need not be set on input.

$y$ (local)

Array, size $(jy-1)*m_y + iy+(m-1)*abs(incy))$ when trans = 'N' or 'n', and $(jy-1)*m_y + iy+(n-1)*abs(incy))$ otherwise.

This array contains the entries of the distributed vector $sub(y)$.

$iy, jy$ (global) The row and column indices in the distributed matrix $Y$ indicating the first row and the first column of the submatrix $sub(y)$, respectively.

descy (global and local) array of dimension 9. The array descriptor of the distributed matrix $Y$.

$incy$ (global) Specifies the increment for the elements of $sub(y)$. Only two values are supported, namely 1 and $m_y$. incy must not be zero.

Output Parameters

$y$ Overwritten by the updated distributed vector $sub(y)$.

**p?ger**

Performs a rank-1 update of a distributed general matrix.
Syntax

void psger (const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incx, float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca);

void pdger (const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy, double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca);

Include Files

• mkl_pblas.h

Description

The p?ger routines perform a distributed matrix-vector operation defined as

\[ \text{sub}(A) := \alpha \text{sub}(x) \text{sub}(y)' + \text{sub}(A), \]

where:

- \( \alpha \) is a scalar,
- \( \text{sub}(A) \) is a \( m \)-by-\( n \) distributed general matrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) is an \( m \)-element distributed vector, \( \text{sub}(y) \) is an \( n \)-element distributed vector,
- \( \text{sub}(x) \) denotes \( X(ix, jx:jx+m-1) \) if \( \text{incx} = m_x \), and \( X(ix: ix+m-1, jx) \) if \( \text{incx} = 1 \),
- \( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy: iy+n-1, jy) \) if \( \text{incy} = 1 \).

Input Parameters

\( m \) (global) Specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( m \geq 0 \).

\( n \) (global) Specifies the number of columns of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

\( \alpha \) (global) Specifies the scalar \( \alpha \).

\( x \) (local) Array, size at least \((jx-1)*m_x + ixt+(m-1)*abs(incx))\). This array contains the entries of the distributed vector \( \text{sub}(x) \).

\( ix, jx \) (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

\( \text{descx} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

\( \text{incy} \) (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.
y
(local) Array, size at least \((jy-1)\cdot m_y + iy+(n-1)\cdot \text{abs(incy)})
This array contains the entries of the distributed vector \(\text{sub}(y)\).

\(iy, jy\)
(global) The row and column indices in the distributed matrix \(Y\) indicating
the first row and the first column of the submatrix \(\text{sub}(y)\), respectively.

descy
(global and local) array of dimension 9. The array descriptor of the
distributed matrix \(Y\).

incy
(global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two
values are supported, namely 1 and \(m_y\). \(incy\) must not be zero.

a
(local) Array, size \((\text{lld}_a, \text{LOCq}(ja+n-1))\).
Before entry this array contains the local pieces of the distributed matrix
\(\text{sub}(A)\).

\(ia, ja\)
(global) The row and column indices in the distributed matrix \(A\) indicating
the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

desca
(global and local) array of dimension 9. The array descriptor of the
distributed matrix \(A\).

Output Parameters

a
Overwritten by the updated distributed matrix \(\text{sub}(A)\).

\textbf{pcgerc}

\textit{Performs a rank-1 update (conjugated) of a
distributed general matrix.}

\textbf{Syntax}

\begin{verbatim}
void pcgerc (const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *ja, const MKL_INT *desca);
void pzgerc (const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *ja, const MKL_INT *desca);
\end{verbatim}

\textbf{Include Files}

- mkl_pblas.h

\textbf{Description}

The \textbf{pcgerc} routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(A) := \alpha \cdot \text{sub}(x) \cdot \text{conjg}(\text{sub}(y)') + \text{sub}(A),
\]
where:

\(\text{alpha}\) is a scalar,

\(\text{sub}(A)\) is a \(m\)-by-\(n\) distributed general matrix, \(\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)\),

\(\text{sub}(x)\) is an \(m\)-element distributed vector, \(\text{sub}(y)\) is an \(n\)-element distributed vector,

\(\text{sub}(x)\) denotes \(X(ix, jx:jx+m-1)\) if \(\text{incx} = m_x\), and \(X(ix: ix+m-1, jx)\) if \(\text{incx} = 1\),

\(\text{sub}(y)\) denotes \(Y(iy, jy:jy+n-1)\) if \(\text{incy} = m_y\), and \(Y(iy: iy+n-1, jy)\) if \(\text{incy} = 1\).

**Input Parameters**

\(m\)  
(global) Specifies the number of rows of the distributed matrix \(\text{sub}(A)\), \(m \geq 0\).

\(n\)  
(global) Specifies the number of columns of the distributed matrix \(\text{sub}(A)\), \(n \geq 0\).

\(\text{alpha}\)  
(global) Specifies the scalar \(\alpha\).

\(x\)  
(local)  
Array, size at least \((jx-1)*m_x + ix+(n-1)*\text{abs(incx)})\).  
This array contains the entries of the distributed vector \(\text{sub}(x)\).

\(ix, jx\)  
(global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix \(\text{sub}(x)\), respectively.

\(\text{descx}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).

\(\text{incx}\)  
(global) Specifies the increment for the elements of \(\text{sub}(x)\). Only two values are supported, namely 1 and \(m_x\). \(\text{incx}\) must not be zero.

\(y\)  
(local)  
Array, size at least \((jy-1)*m_y + iy+(n-1)*\text{abs(incy)})\).  
This array contains the entries of the distributed vector \(\text{sub}(y)\).

\(iy, jy\)  
(global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \(\text{sub}(y)\), respectively.

\(\text{descy}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).

\(\text{incy}\)  
(global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two values are supported, namely 1 and \(m_y\). \(\text{incy}\) must not be zero.

\(a\)  
(local)  
Array, size at least \((\text{ld}_a, \text{LOCq}(ja+n-1))\). Before entry this array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

\(ia, ja\)  
(global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

\(\text{desca}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).
Output Parameters

Overwritten by the updated distributed matrix \( \text{sub}(A) \).

\textbf{pcgeru}

Performs a rank-1 update (unconjugated) of a distributed general matrix.

**Syntax**

```c
void pcgeru (const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_INT *ja, const MKL_INT *desca);
```

```c
void pzgeru (const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_INT *ja, const MKL_INT *desca);
```

**Include Files**

- mkl_pblas.h

**Description**

The p?geru routines perform a matrix-vector operation defined as

\[
\text{sub}(A) := \alpha \text{sub}(x) \ast \text{sub}(y) \ast \text{sub}(A),
\]

where:

- \( \alpha \) is a scalar,
- \( \text{sub}(A) \) is a \( m \)-by-\( n \) distributed general matrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) is an \( m \)-element distributed vector, \( \text{sub}(y) \) is an \( n \)-element distributed vector,
- \( \text{sub}(x) \) denotes \( X(ix, jx:jx+m-1) \) if \( \text{incx} = m \), and \( X(ix: ix+m-1, jx) \) if \( \text{incx} = 1 \),
- \( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m \), and \( Y(iy: iy+n-1, jy) \) if \( \text{incy} = 1 \).

**Input Parameters**

- \( m \) (global) Specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( m \geq 0 \).
- \( n \) (global) Specifies the number of columns of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).
- \( \alpha \) (global)
  Specifies the scalar \( \alpha \).
- \( x \) (local)
  Array, size at least \( (jx-1)*m_x + ix+(n-1)*\text{abs}(\text{incx}) \).
This array contains the entries of the distributed vector \(\text{sub}(x)\).

\(ix, jx\)  
(global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix \(\text{sub}(x)\), respectively.

\(\text{descx}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).

\(\text{incy}\)  
(global) Specifies the increment for the elements of \(\text{sub}(x)\). Only two values are supported, namely 1 and \(m_x\). \(\text{incy}\) must not be zero.

\(y\)  
(local)  
Array, size at least \((jy-1)*m_y + iy+(n-1)*\text{abs}(incy))\). This array contains the entries of the distributed vector \(\text{sub}(y)\).

\(iy, jy\)  
(global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \(\text{sub}(y)\), respectively.

\(\text{descy}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).

\(\text{incy}\)  
(global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two values are supported, namely 1 and \(m_y\). \(\text{incy}\) must not be zero.

\(a\)  
(local)  
Array, size at least \((lld_a, \text{LOCq}(ja+n-1))\). Before entry this array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

\(ia, ja\)  
(global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

\(\text{desca}\)  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).

Output Parameters

\(a\)  
Overwritten by the updated distributed matrix \(\text{sub}(A)\).

\textbf{p?hemv}

\textit{Computes a distributed matrix-vector product using a Hermitian matrix.}

\textbf{Syntax}

```c
void pchemv (const char *uplo , const MKL_INT *n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , const MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );

void pzhemv (const char *uplo , const MKL_INT *n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , const MKL_Complex16 *beta , MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```
Include Files

- mkl_pblas.h

Description

The p?hemv routines perform a distributed matrix-vector operation defined as

\[
\text{sub}\(y\) := alpha*\text{sub}(A)*\text{sub}(x) + beta*\text{sub}(y),
\]

where:

- \(alpha\) and \(beta\) are scalars,
- \(\text{sub}(A)\) is an \(n\)-by-\(n\) Hermitian distributed matrix, \(\text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1)\),
- \(\text{sub}(x)\) and \(\text{sub}(y)\) are distributed vectors.

\(\text{sub}(x)\) denotes \(X(ix, jx:jx+n-1)\) if \(incx = m_x\), and \(X(ix: ix+n-1, jx)\) if \(incx = 1\),
\(\text{sub}(y)\) denotes \(Y(iy, jy: jy+n-1)\) if \(incy = m_y\), and \(Y(iy: iy+n-1, jy)\) if \(incy = 1\).

Input Parameters

**uplo**

(global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix \(\text{sub}(A)\) is used:

- If \(uplo = 'U'\) or \(u\), then the upper triangular part of the \(\text{sub}(A)\) is used.
- If \(uplo = 'L'\) or \(l\), then the lower triangular part of the \(\text{sub}(A)\) is used.

**n**

(global) Specifies the order of the distributed matrix \(\text{sub}(A)\), \(n \geq 0\).

**alpha**

(global) Specifies the scalar \(alpha\).

**a**

(local)

Array, size \((lld_a, LOCq(ja+n-1))\). This array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

Before entry when \(uplo = 'U'\) or \(u\), the \(n\)-by-\(n\) upper triangular part of the distributed matrix \(\text{sub}(A)\) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of \(\text{sub}(A)\) is not referenced, and when \(uplo = 'L'\) or \(l\), the \(n\)-by-\(n\) lower triangular part of the distributed matrix \(\text{sub}(A)\) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of \(\text{sub}(A)\) is not referenced.

**ia, ja**

(global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

**desca**

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).

**x**

(local)

Array, size at least \((jx-1)*m_x + ix+(n-1)*abs(incx))\). This array contains the entries of the distributed vector \(\text{sub}(x)\).
(global) The row and column indices in the distributed matrix $X$ indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

descx
(global and local) array of dimension 9. The array descriptor of the distributed matrix $X$.

incx
(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and $m_x$. incx must not be zero.

beta
(global)
Specifies the scalar beta. When beta is set to zero, then $\text{sub}(y)$ need not be set on input.

y
(local)
Array, size at least $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy))$. This array contains the entries of the distributed vector $\text{sub}(y)$.

iy, jy
(global) The row and column indices in the distributed matrix $Y$ indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.

descy
(global and local) array of dimension 9. The array descriptor of the distributed matrix $Y$.

incy
(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and $m_y$. incy must not be zero.

Output Parameters

$y$
Overwritten by the updated distributed vector $\text{sub}(y)$.

p?ahemv
Computes a distributed matrix-vector product using absolute values for a Hermitian matrix.

Syntax

```c
void pcahemv (const char *uplo , const MKL_INT *n , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , const MKL_Complex8 *beta , MKL_Complex8 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
void pzahemv (const char *uplo , const MKL_INT *n , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , const MKL_Complex16 *beta , MKL_Complex16 *y , const MKL_INT *iy , const MKL_INT *jy , const MKL_INT *descy , const MKL_INT *incy );
```

Include Files

- mkl_pblas.h
Description

The `pahemv` routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(y) := \text{abs}(\alpha) \times \text{abs}(\text{sub}(A)) \times \text{abs}(\text{sub}(x)) + \text{abs}(\beta \times \text{sub}(y)),
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(A) \) is a \( n \times n \) Hermitian distributed matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) and \( \text{sub}(y) \) are distributed vectors.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx} = m_x \), and \( X(ix:ix+n-1, jx) \) if \( \text{incx} = 1 \),

\( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy:iy+n-1, jy) \) if \( \text{incy} = 1 \).

Input Parameters

- **uplo** (global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix \( \text{sub}(A) \) is used:
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(A) \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(A) \) is used.

- **n** (global) Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha** (global) Specifies the scalar \( \alpha \).

- **a** (local)
  - Array, size \((lld_a, \text{LOCq}(ja+n-1))\). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \).
  - Before entry when \( \text{uplo} = 'U' \) or \( 'u' \), the \( n \times n \) upper triangular part of the distributed matrix \( \text{sub}(A) \) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced, and when \( \text{uplo} = 'L' \) or \( 'l' \), the \( n \times n \) lower triangular part of the distributed matrix \( \text{sub}(A) \) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

- **ia, ja** (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

- **desca** (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

- **x** (local)
  - Array, size at least \((jx-1)*m_x + ix+(n-1)*\text{abs(incx)}\).
  - This array contains the entries of the distributed vector \( \text{sub}(x) \).

- **ix, jx** (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

- **descx** (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).
incx
(global) Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and m_x. incx must not be zero.

beta
(global)
Specifies the scalar beta. When beta is set to zero, then sub(y) need not be set on input.

y
(local)
Array, size at least (jy-1)*m_y + iy+(n-1)*abs(incy)). This array contains the entries of the distributed vector sub(y).

iy, jy
(global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix sub(y), respectively.

descy
(global and local) array of dimension 9. The array descriptor of the distributed matrix Y.

incy
(global) Specifies the increment for the elements of sub(y). Only two values are supported, namely 1 and m_y. incy must not be zero.

Output Parameters

y
Overwritten by the updated distributed vector sub(y).

p?her
Performs a rank-1 update of a distributed Hermitian matrix.

Syntax

#include <mkl_pblas.h>

void pcher (const char *uplo, const MKL_INT *n, const float *alpha, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca );

void pzher (const char *uplo, const MKL_INT *n, const double *alpha, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca );

Include Files

• mkl_pblas.h

Description

The p?her routines perform a distributed matrix-vector operation defined as

\[ \text{sub}(A) := \alpha \text{sub}(x) \cdot \text{conjg}(\text{sub}(x))' + \text{sub}(A), \]

where:

\( \alpha \) is a real scalar,

\( \text{sub}(A) \) is a \( n \)-by-\( n \) distributed Hermitian matrix, \( \text{sub}(A)=A(ia:ia+n-1, ja:ja+n-1) \),

\( \text{sub}(x) \) is distributed vector.
sub(x) denotes \( X(ix, jx:jx+n-1) \) if incx = m\_x, and \( X(ix: ix+n-1, jx) \) if incx = 1.

**Input Parameters**

- **uplo** (global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix sub(A) is used:
  - If \( uplo = 'U' \) or \( 'u' \), then the upper triangular part of the sub(A) is used.
  - If \( uplo = 'L' \) or \( 'l' \), then the lower triangular part of the sub(A) is used.

- **n** (global) Specifies the order of the distributed matrix sub(A), \( n \geq 0 \).

- **alpha** (global)
  Specifies the scalar \( \alpha \).

- **x** (local)
  Array, size at least \((jx-1)*m\_x + ix+(n-1)*abs(incx))\).
  This array contains the entries of the distributed vector sub(x).

- **ix, jx** (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix sub(x), respectively.

- **descx** (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

- **incx** (global) Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and \( m\_x \). incx must not be zero.

- **a** (local)
  Array, size \((lld\_a, LOCq(ja+n-1))\). This array contains the local pieces of the distributed matrix sub(A).

  Before entry with \( uplo = 'U' \) or \( 'u' \), the \( n\)-by-\( n \) upper triangular part of the distributed matrix sub(A) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(A) is not referenced, and with \( uplo = 'L' \) or \( 'l' \), the \( n\)-by-\( n \) lower triangular part of the distributed matrix sub(A) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(A) is not referenced.

- **ia, ja** (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix sub(A), respectively.

- **desca** (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

**Output Parameters**

- **a** With \( uplo = 'U' \) or \( 'u' \), the upper triangular part of the array a is overwritten by the upper triangular part of the updated distributed matrix sub(A).
With uplo = 'L' or 'l', the lower triangular part of the array \( a \) is overwritten by the lower triangular part of the updated distributed matrix sub(A).

**p?her2**

Performs a rank-2 update of a distributed Hermitian matrix.

**Syntax**

void pcher2 (const char *uplo, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex8 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca);

void pzher2 (const char *uplo, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const MKL_Complex16 *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca);

**Include Files**

- mkl_pblas.h

**Description**

The p?her2 routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(A) := \alpha \text{sub}(x) \text{conj}(\text{sub}(y))' + \text{conj}(\alpha) \text{sub}(y) \text{conj}(\text{sub}(x))' + \text{sub}(A),
\]

where:

- \( \alpha \) is a scalar,
- \( \text{sub}(A) \) is a \( n \)-by-\( n \) distributed Hermitian matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) and \( \text{sub}(y) \) are distributed vectors.

\( \text{sub}(x) \) denotes \( X(ix, jx:ix+n-1) \) if \( \text{incx} = m_x \), and \( X(ix, ix+n-1, jx) \) if \( \text{incx} = 1 \),

\( \text{sub}(y) \) denotes \( Y(iy, jy:iy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy, iy+n-1, jy) \) if \( \text{incy} = 1 \).

**Input Parameters**

- **uplo**
  
  (global) Specifies whether the upper or lower triangular part of the distributed Hermitian matrix \( \text{sub}(A) \) is used:
  
  If uplo = 'U' or 'u', then the upper triangular part of the \( \text{sub}(A) \) is used.

  If uplo = 'L' or 'l', then the low triangular part of the \( \text{sub}(A) \) is used.

- **n**

  (global)

  Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha**

  (global)

  Specifies the scalar \( \alpha \).
**x**

(local)

Array, size at least \((jx-1)*m_x + ix+(n-1)*abs(incx)\). This array contains the entries of the distributed vector \(\text{sub}(x)\).

**ix, jx**

(global) The row and column indices in the distributed matrix \(X\) indicating the first row and the first column of the submatrix \(\text{sub}(x)\), respectively.

**descx**

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(X\).

**incy**

(global) Specifies the increment for the elements of \(\text{sub}(x)\). Only two values are supported, namely 1 and \(m_x\). \(incx\) must not be zero.

**y**

(local)

Array, size at least \((jy-1)*m_y + iy+(n-1)*abs(incy)\). This array contains the entries of the distributed vector \(\text{sub}(y)\).

**iy, jy**

(global) The row and column indices in the distributed matrix \(Y\) indicating the first row and the first column of the submatrix \(\text{sub}(y)\), respectively.

**descy**

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(Y\).

**incy**

(global) Specifies the increment for the elements of \(\text{sub}(y)\). Only two values are supported, namely 1 and \(m_y\). \(incy\) must not be zero.

**a**

(local)

Array, size \((lld_a, \text{LOCq}(ja+n-1))\). This array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

Before entry with \(\text{uplo} = \text{'}U\text{' or 'u'}\), the \(n\)-by-\(n\) upper triangular part of the distributed matrix \(\text{sub}(A)\) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of \(\text{sub}(A)\) is not referenced, and with \(\text{uplo} = \text{'}L\text{' or 'l'}\), the \(n\)-by-\(n\) lower triangular part of the distributed matrix \(\text{sub}(A)\) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of \(\text{sub}(A)\) is not referenced.

**ia, ja**

(global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

**desca**

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).

**Output Parameters**

**a**

With \(\text{uplo} = \text{'}U\text{' or 'u'}\), the upper triangular part of the array \(a\) is overwritten by the upper triangular part of the updated distributed matrix \(\text{sub}(A)\).

With \(\text{uplo} = \text{'}L\text{' or 'l'}\), the lower triangular part of the array \(a\) is overwritten by the lower triangular part of the updated distributed matrix \(\text{sub}(A)\).
p?symv

Computes a distributed matrix-vector product using a symmetric matrix.

Syntax

void pssmv (const char *uplo, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const float *beta, float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pdsymv (const char *uplo, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, const double *beta, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

Include Files

• mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

\[ \text{sub}(y) := \text{alpha} \times \text{sub}(A) \times \text{sub}(x) + \text{beta} \times \text{sub}(y), \]

where:

\( \alpha \) and \( \beta \) are scalars,

\( \text{sub}(A) \) is a \( n \times n \) symmetric distributed matrix, \( \text{sub}(A)=A(ia:ia+n-1, ja:ja+n-1) \),

\( \text{sub}(x) \) and \( \text{sub}(y) \) are distributed vectors.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( incx = m_x \), and \( X(ix: ix+n-1, jx) \) if \( incx = 1 \),

\( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( incy = m_y \), and \( Y(iy: iy+n-1, jy) \) if \( incy = 1 \).

Input Parameters

- **uplo** (global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \( \text{sub}(A) \) is used:
  
  If \( uplo = 'U' \) or \'u', then the upper triangular part of the \( \text{sub}(A) \) is used.
  
  If \( uplo = 'L' \) or \'l', then the lower triangular part of the \( \text{sub}(A) \) is used.

- **n** (global) Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha** (global) Specifies the scalar \( \alpha \).

- **a** (local) Array, size \( (lld_a, LOCq(ja+n-1)) \). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \).
Before entry when uplo = 'U' or 'u', the n-by-n upper triangular part of the distributed matrix sub(A) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of sub(A) is not referenced, and when uplo = 'L' or 'l', the n-by-n lower triangular part of the distributed matrix sub(A) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of sub(A) is not referenced.

\( \text{ia, ja} \) (global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix sub(A), respectively.

\( \text{desca} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix A.

\( x \) (local)
Array, size at least \((jx-1)*m_x + ix+(n-1)*\text{abs}(incx))\).
This array contains the entries of the distributed vector sub(x).

\( \text{ix, jx} \) (global) The row and column indices in the distributed matrix X indicating the first row and the first column of the submatrix sub(x), respectively.

\( \text{descx} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix X.

\( \text{incx} \) (global) Specifies the increment for the elements of sub(x). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

\( \text{beta} \) (global)
 Specifies the scalar \( \text{beta} \). When \( \text{beta} \) is set to zero, then sub(y) need not be set on input.

\( y \) (local)
Array, size at least \((jy-1)*m_y + iy+(n-1)*\text{abs}(incy))\).
This array contains the entries of the distributed vector sub(y).

\( \text{iy, jy} \) (global) The row and column indices in the distributed matrix Y indicating the first row and the first column of the submatrix sub(y), respectively.

\( \text{descy} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix Y.

\( \text{incy} \) (global) Specifies the increment for the elements of sub(y). Only two values are supported, namely 1 and \( m_y \). \( \text{incy} \) must not be zero.

**Output Parameters**

\( y \) Overwritten by the updated distributed vector sub(y).

**p?asymv**

*Computes a distributed matrix-vector product using absolute values for a symmetric matrix.*
Syntax

void psasymv (const char *uplo, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const float *beta, float *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

void pdasymv (const char *uplo, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const double *beta, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

Include Files

• mkl_pblas.h

Description

The p?symv routines perform a distributed matrix-vector operation defined as

\[ \text{sub}(y) := \text{abs}(\alpha) \cdot \text{abs}(\text{sub}(A)) \cdot \text{abs}(\text{sub}(x)) + \text{abs}(\beta \cdot \text{sub}(y)), \]

where:

\( \alpha \) and \( \beta \) are scalars,

\( \text{sub}(A) \) is a \( n \)-by-\( n \) symmetric distributed matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \),

\( \text{sub}(x) \) and \( \text{sub}(y) \) are distributed vectors.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx} = m_x \), and \( X(ix:ix+n-1, jx) \) if \( \text{incx} = 1 \),

\( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy:iy+n-1, jy) \) if \( \text{incy} = 1 \).

Input Parameters

\( \text{uplo} \)

(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \( \text{sub}(A) \) is used:

If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(A) \) is used.

If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(A) \) is used.

\( \text{n} \)

(global) Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

\( \text{alpha} \)

(global)

Specifies the scalar \( \alpha \).

\( \text{a} \)

(local)

Array, size \( (lld\ a, \text{LOCq}(ja+n-1)) \). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \).

Before entry when \( \text{uplo} = 'U' \) or \( 'u' \), the \( n \)-by-\( n \) upper triangular part of the distributed matrix \( \text{sub}(A) \) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced, and when \( \text{uplo} = 'L' \) or \( 'l' \), the \( n \)-by-\( n \) lower
triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

$ia, ja$  
(global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.

desca  
(global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

$x$  
(local)  
Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx))$.  
This array contains the entries of the distributed vector $\text{sub}(x)$.

$ix, jx$  
(global) The row and column indices in the distributed matrix $X$ indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

descx  
(global and local) array of dimension 9. The array descriptor of the distributed matrix $X$.

$incx$  
(global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and $m_x$. $incx$ must not be zero.

$beta$  
(global)  
Specifies the scalar $beta$. When $beta$ is set to zero, then $\text{sub}(y)$ need not be set on input.

$y$  
(local)  
Array, size at least $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy))$.  
This array contains the entries of the distributed vector $\text{sub}(y)$.

$iy, jy$  
(global) The row and column indices in the distributed matrix $Y$ indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.

descy  
(global and local) array of dimension 9. The array descriptor of the distributed matrix $Y$.

$incy$  
(global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and $m_y$. $incy$ must not be zero.

**Output Parameters**

$y$  
Overwritten by the updated distributed vector $\text{sub}(y)$.

**p?syr**  
**Performs a rank-1 update of a distributed symmetric matrix.**

**Syntax**

```c
void pssyr (const char *uplo , const MKL_INT *n , const float *alpha , const float *x ,
const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx ,
float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca );
```
void pdsyr (const char *uplo, const MKL_INT *n, const double *alpha, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx, double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca);

**Include Files**

- mkl_pblas.h

**Description**

The `p?syr` routines perform a distributed matrix-vector operation defined as

\[ \text{sub}(A) := \alpha \cdot \text{sub}(x) \cdot \text{sub}(x)' + \text{sub}(A), \]

where:

- \( \alpha \) is a scalar,

- \( \text{sub}(A) \) is a \( n \times n \) distributed symmetric matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \),

- \( \text{sub}(x) \) is a distributed vector.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx} = m_x \), and \( X(ix: ix+n-1, jx) \) if \( \text{incx} = 1 \).

**Input Parameters**

- **uplo** (global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \( \text{sub}(A) \) is used:
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(A) \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(A) \) is used.

- **n** (global) Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

- **alpha** (global)
  - Specifies the scalar \( \alpha \).

- **x** (local)
  - Array, size at least \( (jx-1) \cdot m_x + ix+(n-1) \cdot \text{abs}(\text{incx}) \).
  - This array contains the entries of the distributed vector \( \text{sub}(x) \).

- **ix, jx** (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

- **descx** (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

- **incx** (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

- **a** (local)
  - Array, size \( (\text{lld}_a, \text{LOCq}(ja+n-1)) \). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \).
  - Before entry with \( \text{uplo} = 'U' \) or \( 'u' \), the \( n \times n \) upper triangular part of the distributed matrix \( \text{sub}(A) \) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced, and with \( \text{uplo} = 'L' \) or \( 'l' \), the \( n \times n \) lower
triangular part of the distributed matrix \( \text{sub}(A) \) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

\( ia, ja \) (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

**Output Parameters**

\( a \)

With \( \text{uplo} = 'U' \) or \( 'u' \), the upper triangular part of the array \( a \) is overwritten by the upper triangular part of the updated distributed matrix \( \text{sub}(A) \).

With \( \text{uplo} = 'L' \) or \( 'l' \), the lower triangular part of the array \( a \) is overwritten by the lower triangular part of the updated distributed matrix \( \text{sub}(A) \).

**p?syr2**

*Performs a rank-2 update of a distributed symmetric matrix.*

**Syntax**

```c
void pssyr2 (const char *uplo , const MKL_INT *n , const float *alpha , const float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca );
void pdsyr2 (const char *uplo , const MKL_INT *n , const double *alpha , const double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incy , double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca );
```

**Include Files**

- mkl_pblas.h

**Description**

The \( p\text{syr2} \) routines perform a distributed matrix-vector operation defined as

\[
\text{sub}(A) := \alpha \text{sub}(x) \text{sub}(y)' + \alpha \text{sub}(y) \text{sub}(x)' + \text{sub}(A),
\]

where:

- \( \alpha \) is a scalar,
- \( \text{sub}(A) \) is a \( n \times n \) distributed symmetric matrix, \( \text{sub}(A)=A(ia:ia+n-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) and \( \text{sub}(y) \) are distributed vectors.
- \( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( \text{incx} = m_x \), and \( X(ix: ix+n-1, jx) \) if \( \text{incx} = 1 \),
- \( \text{sub}(y) \) denotes \( Y(iy, jy:jy+n-1) \) if \( \text{incy} = m_y \), and \( Y(iy: iy+n-1, jy) \) if \( \text{incy} = 1 \).
Input Parameters

uplo (global) Specifies whether the upper or lower triangular part of the distributed symmetric matrix $\text{sub}(A)$ is used:

- If $\text{uplo} = 'U'$ or 'u', then the upper triangular part of the $\text{sub}(A)$ is used.
- If $\text{uplo} = 'L'$ or 'l', then the lower triangular part of the $\text{sub}(A)$ is used.

$n$ (global) Specifies the order of the distributed matrix $\text{sub}(A)$, $n \geq 0$.

$\alpha$ (global) Specifies the scalar $\alpha$.

$x$ (local)
Array, size at least $(jx-1)*m_x + ix+(n-1)*\text{abs}(incx))$. This array contains the entries of the distributed vector $\text{sub}(x)$.

$ix, jx$ (global) The row and column indices in the distributed matrix $X$ indicating the first row and the first column of the submatrix $\text{sub}(x)$, respectively.

descx (global and local) array of dimension 9. The array descriptor of the distributed matrix $X$.

$incx$ (global) Specifies the increment for the elements of $\text{sub}(x)$. Only two values are supported, namely 1 and $m_x$. $incx$ must not be zero.

$y$ (local)
Array, size at least $(jy-1)*m_y + iy+(n-1)*\text{abs}(incy))$. This array contains the entries of the distributed vector $\text{sub}(y)$.

$iy, jy$ (global) The row and column indices in the distributed matrix $Y$ indicating the first row and the first column of the submatrix $\text{sub}(y)$, respectively.

descy (global and local) array of dimension 9. The array descriptor of the distributed matrix $Y$.

$incy$ (global) Specifies the increment for the elements of $\text{sub}(y)$. Only two values are supported, namely 1 and $m_y$. $incy$ must not be zero.

$a$ (local)
Array, size $(lld_a, \text{LOCq}(ja+n-1))$. This array contains the local pieces of the distributed matrix $\text{sub}(A)$.

Before entry with $\text{uplo} = 'U'$ or 'u', the $n$-by-$n$ upper triangular part of the distributed matrix $\text{sub}(A)$ must contain the upper triangular part of the distributed symmetric matrix and the strictly lower triangular part of $\text{sub}(A)$ is not referenced, and with $\text{uplo} = 'L'$ or 'l', the $n$-by-$n$ lower triangular part of the distributed matrix $\text{sub}(A)$ must contain the lower triangular part of the distributed symmetric matrix and the strictly upper triangular part of $\text{sub}(A)$ is not referenced.

$ia, ja$ (global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix $\text{sub}(A)$, respectively.
desca

(global and local) array of dimension 9. The array descriptor of the distributed matrix A.

Output Parameters

a

With uplo = 'U' or 'u', the upper triangular part of the array a is overwritten by the upper triangular part of the updated distributed matrix sub(A).

With uplo = 'L' or 'l', the lower triangular part of the array a is overwritten by the lower triangular part of the updated distributed matrix sub(A).

p?trmv

Computes a distributed matrix-vector product using a triangular matrix.

Syntax

void pstrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT *n , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , float *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pdtrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT *n , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , double *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pctrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT *n , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , MKL_Complex8 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

void pztrmv (const char *uplo , const char *trans , const char *diag , const MKL_INT *n , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , MKL_Complex16 *x , const MKL_INT *ix , const MKL_INT *jx , const MKL_INT *descx , const MKL_INT *incx );

Include Files

- mkl_pblas.h

Description

The p?trmv routines perform one of the following distributed matrix-vector operations defined as

\( \text{sub}(x) := \text{sub}(A) * \text{sub}(x), \text{or sub}(x) := \text{sub}(A)'^* \text{sub}(x), \text{or sub}(x) := \text{conjg}(\text{sub}(A))'* \text{sub}(x), \)

where:

\( \text{sub}(A) \) is a \( n \)-by-\( n \) unit, or non-unit, upper or lower triangular distributed matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1), \)

\( \text{sub}(x) \) is an \( n \)-element distributed vector.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( incx = m_x \), and \( X(ix: ix+n-1, jx) \) if \( incx = 1, \)
Input Parameters

**uplo**  
(global) Specifies whether the distributed matrix \( \text{sub}(A) \) is upper or lower triangular:
- if \( \text{uplo} = 'U' \) or \( 'u' \), then the matrix is upper triangular;
- if \( \text{uplo} = 'L' \) or \( 'l' \), then the matrix is lower triangular.

**trans**  
(global) Specifies the form of \( \text{op}(\text{sub}(A)) \) used in the matrix equation:
- if \( \text{transa} = 'N' \) or \( 'n' \), then \( \text{sub}(x) := \text{sub}(A) \ast \text{sub}(x) \);
- if \( \text{transa} = 'T' \) or \( 't' \), then \( \text{sub}(x) := \text{sub}(A)'^* \text{sub}(x) \);
- if \( \text{transa} = 'C' \) or \( 'c' \), then \( \text{sub}(x) := \text{conjg}(\text{sub}(A)') \ast \text{sub}(x) \).

**diag**  
(global) Specifies whether the matrix \( \text{sub}(A) \) is unit triangular:
- if \( \text{diag} = 'U' \) or \( 'u' \), then the matrix is unit triangular;
- if \( \text{diag} = 'N' \) or \( 'n' \), then the matrix is not unit triangular.

**n**  
(global) Specifies the order of the distributed matrix \( \text{sub}(A) \), \( n \geq 0 \).

**a**  
(local)  
Array, size at least \((\text{lld}_a, \text{LOCq}(1, ja+n-1))\).

Before entry with \( \text{uplo} = 'U' \) or \( 'u' \), this array contains the local entries corresponding to the entries of the upper triangular distributed matrix \( \text{sub}(A) \), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix \( \text{sub}(A) \) is not referenced.

Before entry with \( \text{uplo} = 'L' \) or \( 'l' \), this array contains the local entries corresponding to the entries of the lower triangular distributed matrix \( \text{sub}(A) \), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix \( \text{sub}(A) \) is not referenced.

When \( \text{diag} = 'U' \) or \( 'u' \), the local entries corresponding to the diagonal elements of the submatrix \( \text{sub}(A) \) are not referenced either, but are assumed to be unity.

**ia, ja**  
(global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

**desca**  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

**x**  
(local)  
Array, size at least \((jx-1) \ast m_x + ix+(n-1) \ast \text{abs}(incx))\).

This array contains the entries of the distributed vector \( \text{sub}(x) \).

**ix, jx**  
(global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

**descx**  
(global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

**incx**  
(global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.
Computes a distributed matrix-vector product using absolute values for a triangular matrix.

**Syntax**

void psatrmv (const char *uplo, const char *trans, const char *diag, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incy, const double *beta, double *y, const MKL_INT *iy, const MKL_INT *jy, const MKL_INT *descy, const MKL_INT *incy);

**Description**

The `psatrmv` routines perform one of the following distributed matrix-vector operations defined as

\[ \text{sub}(y) := \text{abs}(\alpha) \times \text{abs}(\text{sub}(A)) \times \text{abs}(\text{sub}(x)) + \text{abs}(\beta \times \text{sub}(y)), \text{or} \]

\[ \text{sub}(y) := \text{abs}(\alpha) \times \text{abs}(\text{sub}(A)^T) \times \text{abs}(\text{sub}(x)) + \text{abs}(\beta \times \text{sub}(y)), \text{or} \]

\[ \text{sub}(y) := \text{abs}(\alpha) \times \text{abs}(\text{conjg}(\text{sub}(A)^T)) \times \text{abs}(\text{sub}(x)) + \text{abs}(\beta \times \text{sub}(y)), \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(A) \) is a \( n \times n \) unit, or non-unit, upper or lower triangular distributed matrix, \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \),
- \( \text{sub}(x) \) is an \( n \)-element distributed vector.

\( \text{sub}(x) \) denotes \( X(ix, jx:jx+n-1) \) if \( incx = m_x \), and \( X(ix: ix+n-1, jx) \) if \( incx = 1 \).
Input Parameters

**uplo**

(global) Specifies whether the distributed matrix sub(A) is upper or lower triangular:

- If `uplo = 'U'` or `'u'`, then the matrix is upper triangular;
- If `uplo = 'L'` or `'l'`, then the matrix is lower triangular.

**trans**

(global) Specifies the form of $\text{op}(\text{sub}(A))$ used in the matrix equation:

- If `trans = 'N'` or `'n'`, then $\text{sub}(y) := |\alpha||\text{sub}(A)||\text{sub}(x)| + |\beta|\text{sub}(y)$;
- If `trans = 'T'` or `'t'`, then $\text{sub}(y) := |\alpha||\text{sub}(A)'||\text{sub}(x)| + |\beta|\text{sub}(y)$;
- If `trans = 'C'` or `'c'`, then $\text{sub}(y) := |\alpha||\text{conjg}(\text{sub}(A)'')||\text{sub}(x)| + |\beta|\text{sub}(y)$.

**diag**

(global) Specifies whether the matrix sub(A) is unit triangular:

- If `diag = 'U'` or `'u'`, then the matrix is unit triangular;
- If `diag = 'N'` or `'n'`, then the matrix is not unit triangular.

**n**

(global) Specifies the order of the distributed matrix sub(A), $n \geq 0$.

**alpha**

(global)

Specifies the scalar $\alpha$.

**a**

(local)

Array, size at least $(\text{lld}_a, \text{LOCq}(1, ja+n-1))$.

Before entry with `uplo = 'U'` or `'u'`, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix sub(A), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix sub(A) is not referenced.

Before entry with `uplo = 'L'` or `'l'`, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(A), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(A) is not referenced.

When `diag = 'U'` or `'u'`, the local entries corresponding to the diagonal elements of the submatrix sub(A) are not referenced either, but are assumed to be unity.

**ia, ja**

(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix sub(A), respectively.

**desca**

(global and local) array of dimension 9. The array descriptor of the distributed matrix A.

**x**

(local)

Array, size at least $(jx-1)*m_x + ixt(n-1)*\text{abs}(incx))$.

This array contains the entries of the distributed vector sub(x).
ix, jx  (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(x) \), respectively.

descx  (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

incx  (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). incx must not be zero.

beta  (global) Specifies the scalar \( \beta \). When \( \beta \) is set to zero, then \( \text{sub}(y) \) need not be set on input.

y  (local) Array, size \((jy-1)*m_y + iy+(m-1)*\text{abs}(incy)\) when \( \text{trans} = 'N' \) or '\n', and \((jy-1)*m_y + iy+(n-1)*\text{abs}(incy)\) otherwise.

This array contains the entries of the distributed vector \( \text{sub}(y) \).

iy, jy  (global) The row and column indices in the distributed matrix \( Y \) indicating the first row and the first column of the submatrix \( \text{sub}(y) \), respectively.

descy  (global and local) array of dimension 9. The array descriptor of the distributed matrix \( Y \).

incy  (global) Specifies the increment for the elements of \( \text{sub}(y) \). Only two values are supported, namely 1 and \( m_y \). incy must not be zero.

Output Parameters

y  Overwritten by the transformed distributed vector \( \text{sub}(y) \).

\textbf{p?trsv}  
Solves a system of linear equations whose coefficients are in a distributed triangular matrix.

\textbf{Syntax}

void pstrsv (const char *uplo, const char *trans, const char *diag, const MKL_INT *n, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, float *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

void pdtrsv (const char *uplo, const char *trans, const char *diag, const MKL_INT *n, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, double *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

void pctrsv (const char *uplo, const char *trans, const char *diag, const MKL_INT *n, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex8 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);
void pztrsv (const char *uplo, const char *trans, const char *diag, const MKL_INT *n, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex16 *x, const MKL_INT *ix, const MKL_INT *jx, const MKL_INT *descx, const MKL_INT *incx);

Include Files
• mkl_pblas.h

Description
The p?trsv routines solve one of the systems of equations:
\[ \text{sub}(A) \times \text{sub}(x) = b, \text{ or } \text{sub}(A)' \times \text{sub}(x) = b, \text{ or } \text{conjg}(\text{sub}(A)') \times \text{sub}(x) = b, \]
where:
\[ \text{sub}(A) \text{ is a } n\times n \text{ unit, or non-unit, upper or lower triangular distributed matrix, } \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1), \]
\[ b \text{ and } \text{sub}(x) \text{ are } n\text{-element distributed vectors,} \]
\[ \text{sub}(x) \text{ denotes } X(ix, jx:jx+n-1) \text{ if } incx = m_x, \text{ and } X(ix: ix+n-1, jx) \text{ if } incx = 1. \]
The routine does not test for singularity or near-singularity. Such tests must be performed before calling this routine.

Input Parameters

uplo
(global) Specifies whether the distributed matrix \text{sub}(A) is upper or lower triangular:
if \text{uplo} = 'U' or 'u', then the matrix is upper triangular;
if \text{uplo} = 'L' or 'l', then the matrix is lower triangular.

trans
(global) Specifies the form of the system of equations:
if \text{transa} = 'N' or 'n', then \text{sub}(A) \times \text{sub}(x) = b;
if \text{transa} = 'T' or 't', then \text{sub}(A)' \times \text{sub}(x) = b;
if \text{transa} = 'C' or 'c', then \text{conjg}(\text{sub}(A)') \times \text{sub}(x) = b.

diag
(global) Specifies whether the matrix \text{sub}(A) is unit triangular:
if \text{diag} = 'U' or 'u' then the matrix is unit triangular;
if \text{diag} = 'N' or 'n', then the matrix is not unit triangular.

n
(global) Specifies the order of the distributed matrix \text{sub}(A), n \geq 0.

a
(local)
Array, size at least (lld_a, LOCq(1, ja+n-1)).

Before entry with \text{uplo} = 'U' or 'u', this array contains the local entries corresponding to the entries of the upper triangular distributed matrix \text{sub}(A), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix \text{sub}(A) is not referenced.

Before entry with \text{uplo} = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix \text{sub}(A), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix \text{sub}(A) is not referenced.
When \( \text{diag} = 'U' \) or \( 'u' \), the local entries corresponding to the diagonal elements of the submatrix \( \text{sub}(A) \) are not referenced either, but are assumed to be unity.

\( \text{ia}, \text{ja} \) (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

\( x \) (local) Array, size at least \((jx-1)\cdot m_x + ix + (n-1)\cdot \text{abs}(\text{incx})\).

This array contains the entries of the distributed vector \( \text{sub}(x) \). Before entry, \( \text{sub}(x) \) must contain the \( n \)-element right-hand side distributed vector \( b \).

\( \text{ix}, \text{jx} \) (global) The row and column indices in the distributed matrix \( X \) indicating the first row and the first column of the submatrix \( \text{sub}(X) \), respectively.

\( \text{descx} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( X \).

\( \text{incx} \) (global) Specifies the increment for the elements of \( \text{sub}(x) \). Only two values are supported, namely 1 and \( m_x \). \( \text{incx} \) must not be zero.

Output Parameters

\( x \) Overwritten with the solution vector.

**PBLAS Level 3 Routines**

The PBLAS Level 3 routines perform distributed matrix-matrix operations. Table "PBLAS Level 3 Routine Groups and Their Data Types" lists the PBLAS Level 3 routine groups and the data types associated with them.

**PBLAS Level 3 Routine Groups and Their Data Types**

<table>
<thead>
<tr>
<th>Routine Group</th>
<th>Data Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{p?geadd} )</td>
<td>( s, d, c, z )</td>
<td>Distributed matrix-matrix sum of general matrices</td>
</tr>
<tr>
<td>( \text{p?tradd} )</td>
<td>( s, d, c, z )</td>
<td>Distributed matrix-matrix sum of triangular matrices</td>
</tr>
<tr>
<td>( \text{p?gemm} )</td>
<td>( s, d, c, z )</td>
<td>Distributed matrix-matrix product of general matrices</td>
</tr>
<tr>
<td>( \text{p?hemm} )</td>
<td>( c, z )</td>
<td>Distributed matrix-matrix product, one matrix is Hermitian</td>
</tr>
<tr>
<td>( \text{p?herk} )</td>
<td>( c, z )</td>
<td>Rank-k update of a distributed Hermitian matrix</td>
</tr>
<tr>
<td>( \text{p?her2k} )</td>
<td>( c, z )</td>
<td>Rank-2k update of a distributed Hermitian matrix</td>
</tr>
<tr>
<td>( \text{p?symm} )</td>
<td>( s, d, c, z )</td>
<td>Matrix-matrix product of distributed symmetric matrices</td>
</tr>
<tr>
<td>( \text{p?syrk} )</td>
<td>( s, d, c, z )</td>
<td>Rank-k update of a distributed symmetric matrix</td>
</tr>
<tr>
<td>( \text{p?syr2k} )</td>
<td>( s, d, c, z )</td>
<td>Rank-2k update of a distributed symmetric matrix</td>
</tr>
</tbody>
</table>
### p?geadd

Performs sum operation for two distributed general matrices.

#### Syntax

```c
void psgeadd (const char *trans, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *beta, float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pdgeadd (const char *trans, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *beta, double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pcgeadd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pzgeadd (const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```

#### Description

The `p?geadd` routines perform sum operation for two distributed general matrices. The operation is defined as

\[
\text{sub}(C) := \beta \text{sub}(C) + \alpha \text{op}\left(\text{sub}(A)\right),
\]

where:

- `op(x)` is one of `op(x) = x`, or `op(x) = x^\ast`,
- `alpha` and `beta` are scalars,
- `sub(C)` is an `m`-by-`n` distributed matrix, `sub(C) = C(ic:ic+m-1, jc:jc+n-1)`.
- `sub(A)` is a distributed matrix, `sub(A) = A(ia:ia+n-1, ja:ja+m-1)`.
Input Parameters

\[\text{trans}\]

(global) Specifies the operation:

- if \(\text{trans} = 'N'\) or \(n\), then \(\text{op}(\text{sub}(A)) := \text{sub}(A)\);
- if \(\text{trans} = 'T'\) or \(t\), then \(\text{op}(\text{sub}(A)) := \text{sub}(A)^\text{T}\);
- if \(\text{trans} = 'C'\) or \(c\), then \(\text{op}(\text{sub}(A)) := \text{sub}(A)^\text{C}\).

\[\text{m}\]

(global) Specifies the number of rows of the distributed matrix \(\text{sub}(C)\) and the number of columns of the submatrix \(\text{sub}(A)\), \(m \geq 0\).

\[\text{n}\]

(global) Specifies the number of columns of the distributed matrix \(\text{sub}(C)\) and the number of rows of the submatrix \(\text{sub}(A)\), \(n \geq 0\).

\[\text{alpha}\]

(global)
Specifies the scalar \(\alpha\).

\[\text{a}\]

(local)
Array, size \((\text{lld}_a, \text{LOCq}(ja+m-1))\). This array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

\[\text{ia}, \text{ja}\]

(global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

\[\text{desc}_{a}\]

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).

\[\text{beta}\]

(global)
Specifies the scalar \(\beta\).
When \(\beta\) is equal to zero, then \(\text{sub}(C)\) need not be set on input.

\[\text{c}\]

(local)
Array, size \((\text{lld}_c, \text{LOCq}(jc+n-1))\).
This array contains the local pieces of the distributed matrix \(\text{sub}(C)\).

\[\text{ic}, \text{jc}\]

(global) The row and column indices in the distributed matrix \(C\) indicating the first row and the first column of the submatrix \(\text{sub}(C)\), respectively.

\[\text{descc}\]

(global and local) array of dimension 9. The array descriptor of the distributed matrix \(C\).

Output Parameters

\[\text{c}\]

Overwritten by the updated submatrix.

\[\text{p?tradd}\]

Performs sum operation for two distributed triangular matrices.
void pstradd (const char *uplo, const char *trans, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *beta, float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pdtradd (const char *uplo, const char *trans, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *beta, double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pctradd (const char *uplo, const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pztradd (const char *uplo, const char *trans, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

Include Files

• mkl_pblas.h

Description

The p?tradd routines perform sum operation for two distributed triangular matrices. The operation is defined as

\[ \text{sub}(C) := \beta \cdot \text{sub}(C) + \alpha \cdot \text{op}(\text{sub}(A)), \]

where:

\( \text{op}(x) \) is one of \( \text{op}(x) = x, \text{or } \text{op}(x) = x', \text{or } \text{op}(x) = \text{conjg}(x') \).

\( \alpha \) and \( \beta \) are scalars,

\( \text{sub}(C) \) is an \( m \)-by-\( n \) distributed matrix, \( \text{sub}(C) = C(\text{ic}:\text{ic}+m-1, \text{jc}:\text{jc}+n-1) \).

\( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+m-1) \).

Input Parameters

\text{uplo} \hspace{2cm} \text{(global)} \text{Specifies whether the distributed matrix } \text{sub}(C) \text{ is upper or lower triangular:}

if \( \text{uplo} = 'U' \text{ or } 'u' \), then the matrix is upper triangular;

if \( \text{uplo} = 'L' \text{ or } 'l' \), then the matrix is lower triangular.

\text{trans} \hspace{2cm} \text{(global)} \text{Specifies the operation:}

if \( \text{trans} = 'N' \text{ or } 'n' \), then \( \text{op}(\text{sub}(A)) := \text{sub}(A) \);

if \( \text{trans} = 'T' \text{ or } 't' \), then \( \text{op}(\text{sub}(A)) := \text{sub}(A)' \);

if \( \text{trans} = 'C' \text{ or } 'c' \), then \( \text{op}(\text{sub}(A)) := \text{conjg}(\text{sub}(A)) \).

\text{m} \hspace{2cm} \text{(global)} \text{Specifies the number of rows of the distributed matrix } \text{sub}(C) \text{ and the number of columns of the submatrix } \text{sub}(A), m \geq 0.
n (global) Specifies the number of columns of the distributed matrix $sub(C)$ and the number of rows of the submatrix $sub(A)$, $n \geq 0$.

alpha (global)
Specifies the scalar $alpha$.

a (local)
Array, size $(lld_a, LOCq(ja+m-1))$. This array contains the local pieces of the distributed matrix $sub(A)$.

ia, ja (global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix $sub(A)$, respectively.

desca (global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

beta (global)
Specifies the scalar $beta$.
When $beta$ is equal to zero, then $sub(C)$ need not be set on input.

c (local)
Array, size $(lld_c, LOCq(jc+n-1))$.
This array contains the local pieces of the distributed matrix $sub(C)$.

ic, jc (global) The row and column indices in the distributed matrix $C$ indicating the first row and the first column of the submatrix $sub(C)$, respectively.

descc (global and local) array of dimension 9. The array descriptor of the distributed matrix $C$.

Output Parameters

c Overwritten by the updated submatrix.

**p?gemm**

*Computes a scalar-matrix-matrix product and adds the result to a scalar-matrix product for distributed matrices.*

**Syntax**

```c
void psgemm (const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const float *beta, const float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pdgemm (const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const double *beta, const double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```
void pcgemm (const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pzgemm (const char *transa, const char *transb, const MKL_INT *m, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

Include Files

- mkl_pblas.h

Description

The p?gemm routines perform a matrix-matrix operation with general distributed matrices. The operation is defined as

\[
\text{sub}(C) := \alpha \cdot \text{op}(\text{sub}(A)) \cdot \text{op}(\text{sub}(B)) + \beta \cdot \text{sub}(C),
\]

where:

- \( \text{op}(x) \) is one of \( \text{op}(x) = x \) or \( \text{op}(x) = x' \).
- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+k-1) \), \( \text{sub}(B) = B(ib:ib+k-1, jb:jb+n-1) \), and \( \text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1) \), are distributed matrices.

Input Parameters

- \textit{transa} (global) Specifies the form of \( \text{op}(\text{sub}(A)) \) used in the matrix multiplication:
  - if \( \text{transa} = 'N' \) or \( 'n' \), then \( \text{op}(\text{sub}(A)) = \text{sub}(A) \);
  - if \( \text{transa} = 'T' \) or \( 't' \), then \( \text{op}(\text{sub}(A)) = \text{sub}(A)' \);
  - if \( \text{transa} = 'C' \) or \( 'c' \), then \( \text{op}(\text{sub}(A)) = \text{sub}(A)' \).

- \textit{transb} (global) Specifies the form of \( \text{op}(\text{sub}(B)) \) used in the matrix multiplication:
  - if \( \text{transb} = 'N' \) or \( 'n' \), then \( \text{op}(\text{sub}(B)) = \text{sub}(B) \);
  - if \( \text{transb} = 'T' \) or \( 't' \), then \( \text{op}(\text{sub}(B)) = \text{sub}(B)' \);
  - if \( \text{transb} = 'C' \) or \( 'c' \), then \( \text{op}(\text{sub}(B)) = \text{sub}(B)' \).

- \textit{m} (global) Specifies the number of rows of the distributed matrices \( \text{op}(\text{sub}(A)) \) and \( \text{sub}(C) \), \( m \geq 0 \).

- \textit{n} (global) Specifies the number of columns of the distributed matrices \( \text{op}(\text{sub}(B)) \) and \( \text{sub}(C) \), \( n \geq 0 \).
  
  The value of \( n \) must be at least zero.

- \textit{k} (global) Specifies the number of columns of the distributed matrix \( \text{op}(\text{sub}(A)) \) and the number of rows of the distributed matrix \( \text{op}(\text{sub}(B)) \).
  
  The value of \( k \) must be greater than or equal to 0.
Specifies the scalar \( \alpha \).

When \( \alpha \) is equal to zero, then the local entries of the arrays \( a \) and \( b \) corresponding to the entries of the submatrices \( \text{sub}(A) \) and \( \text{sub}(B) \) respectively need not be set on input.

Array, size \( \text{lld}_a \times \text{kla} \), where \( \text{kla} = \text{LOCc}(ja+k-1) \) when \( \text{transa} = 'N' \) or \( 'n' \), and is \( \text{LOCq}(ja+m-1) \) otherwise. Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(A) \).

The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively

(global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

Array, size \( \text{lld}_b \times \text{klb} \), where \( \text{klb} = \text{LOCc}(jb+n-1) \) when \( \text{transb} = 'N' \) or \( 'n' \), and is \( \text{LOCq}(jb+k-1) \) otherwise. Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(B) \).

The row and column indices in the distributed matrix \( B \) indicating the first row and the first column of the submatrix \( \text{sub}(B) \), respectively

(global and local) array of dimension 9. The array descriptor of the distributed matrix \( B \).

Specifies the scalar \( \beta \).

When \( \beta \) is equal to zero, then \( \text{sub}(C) \) need not be set on input.

Array, size \( (\text{lld}_a, \text{LOCq}(jc+n-1)) \). Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(C) \).

The row and column indices in the distributed matrix \( C \) indicating the first row and the first column of the submatrix \( \text{sub}(C) \), respectively

(global and local) array of dimension 9. The array descriptor of the distributed matrix \( C \).

Output Parameters

Overwritten by the \( m \times n \) distributed matrix
\[
\alpha \times \text{op}(\text{sub}(A)) \times \text{op}(\text{sub}(B)) + \beta \times \text{sub}(C).
\]

Performs a scalar-matrix-matrix product (one matrix operand is Hermitian) and adds the result to a scalar-matrix product.
Syntax

```c
void pchemm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pzhemm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex16 *beta, const MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```

Include Files

- mkl_pblasc.h

Description

The p?hemm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

\[
\text{sub}(C):= \alpha \text{sub}(A) \text{sub}(B) + \beta \text{sub}(C),
\]

or

\[
\text{sub}(C):= \alpha \text{sub}(B) \text{sub}(A) + \beta \text{sub}(C),
\]

where:

- \(\alpha\) and \(\beta\) are scalars,
- \(\text{sub}(A)\) is a Hermitian distributed matrix, \(\text{sub}(A)= A(ia:ia+m-1, ja:ja+m-1)\), if \(\text{side} = 'L'\), and \(\text{sub}(A)= A(ia:ia+n-1, ja:ja+n-1)\), if \(\text{side} = 'R'\).
- \(\text{sub}(B)\) and \(\text{sub}(C)\) are \(m\)-by-\(n\) distributed matrices.
- \(\text{sub}(B)= B(ib:ib+m-1, jb:jb+n-1)\), \(\text{sub}(C)= C(ic:ic+m-1, jc:jc+n-1)\).

Input Parameters

- **side** (global) Specifies whether the Hermitian distributed matrix \(\text{sub}(A)\) appears on the left or right in the operation:
  
  - if \(\text{side} = 'L'\) or \('l'\), then \(\text{sub}(C):= \alpha \text{sub}(A) \text{sub}(B) + \beta \text{sub}(C)\);
  
  - if \(\text{side} = 'R'\) or \('r'\), then \(\text{sub}(C):= \alpha \text{sub}(B) \text{sub}(A) + \beta \text{sub}(C)\).

- **uplo** (global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix \(\text{sub}(A)\) is used:
  
  - if \(\text{uplo} = 'U'\) or \('u'\), then the upper triangular part is used;
  
  - if \(\text{uplo} = 'L'\) or \('l'\), then the lower triangular part is used.

- **m** (global) Specifies the number of rows of the distribute submatrix \(\text{sub}(C)\), \(m \geq 0\).

- **n** (global) Specifies the number of columns of the distribute submatrix \(\text{sub}(C)\), \(n \geq 0\).
alpha

Specifies the scalar alpha.

a

Array, size (lld_a, LOCq(ja+na-1)). Before entry this array must contain the local pieces of the symmetric distributed matrix sub(A), such that when uplo = 'U' or 'u', the na-by-na upper triangular part of the distributed matrix sub(A) must contain the upper triangular part of the Hermitian distributed matrix and the strictly lower triangular part of sub(A) is not referenced, and when uplo = 'L' or 'l', the na-by-na lower triangular part of the distributed matrix sub(A) must contain the lower triangular part of the Hermitian distributed matrix and the strictly upper triangular part of sub(A) is not referenced.

ia, ja

(global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix sub(A), respectively.

desca

(global and local) array of dimension 9. The array descriptor of the distributed matrix A.

b

Array, size (lld_b, LOCq(jb+n-1)). Before entry this array must contain the local pieces of the distributed matrix sub(B).

ib, jb

(global) The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix sub(B), respectively.

descb

(global and local) array of dimension 9. The array descriptor of the distributed matrix B.

beta

Specifies the scalar beta. When beta is set to zero, then sub(C) need not be set on input.

c

Array, size (lld_c, LOCq(jc+n-1)). Before entry this array must contain the local pieces of the distributed matrix sub(C).

ic, jc

(global) The row and column indices in the distributed matrix C indicating the first row and the first column of the submatrix sub(C), respectively.

descC

(global and local) array of dimension 9. The array descriptor of the distributed matrix C.

Output Parameters

c

Overwritten by the m-by-n updated distributed matrix.

p?herk

Performs a rank-k update of a distributed Hermitian matrix.
Syntax

```c
void pcherk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const float *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *beta , MKL_Complex8 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
void pzherk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const double *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const double *beta , MKL_Complex16 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- mkl_pblas.h

Description

The `p?herk` routines perform a distributed matrix-matrix operation defined as

\[
\text{sub}(C) := \alpha \text{sub}(A) \ast \text{conjg}(\text{sub}(A))' + \beta \text{sub}(C),
\]

or

\[
\text{sub}(C) := \alpha \text{conjg}(\text{sub}(A))' \ast \text{sub}(A) + \beta \text{sub}(C),
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(C) \) is an \( n \)-by-\( n \) Hermitian distributed matrix, \( \text{sub}(C) = C(\text{ic:ic}+n-1, \text{jc:jc}+n-1) \).
- \( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia:ia}+n-1, \text{ja:ja}+k-1) \), if \( \text{trans} = 'N' \) or \( 'n' \), and \( \text{sub}(A) = A(\text{ia:ia}+k-1, \text{ja:ja}+n-1) \) otherwise.

Input Parameters

- \( \text{uplo} \) (global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix \( \text{sub}(C) \) is used:
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(C) \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(C) \) is used.
- \( \text{trans} \) (global) Specifies the operation:
  - If \( \text{trans} = 'N' \) or \( 'n' \), then \( \text{sub}(C) := \alpha \text{sub}(A) \ast \text{conjg}(\text{sub}(A))' + \beta \text{sub}(C) \);
  - If \( \text{trans} = 'C' \) or \( 'c' \), then \( \text{sub}(C) := \alpha \text{conjg}(\text{sub}(A))' \ast \text{sub}(A) + \beta \text{sub}(C) \).
- \( n \) (global) Specifies the order of the distributed matrix \( \text{sub}(C) \), \( n \geq 0 \).
- \( k \) (global) On entry with \( \text{trans} = 'N' \) or \( 'n' \), \( k \) specifies the number of columns of the distributed matrix \( \text{sub}(A) \), and on entry with \( \text{trans} = 'T' \) or \( 't' \) or \( 'C' \) or \( 'c' \), \( k \) specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( k \geq 0 \).
- \( \alpha \) (global) Specifies the scalar \( \alpha \).
Array, size \((lld_a, kla)\), where \(kla\) is \(\text{LOCq}(ja+k-1)\) when \(\text{trans} = \text{'N'}\) or \text{'n'}, and is \(\text{LOCq}(ja+n-1)\) otherwise. Before entry with \(\text{trans} = \text{'N'}\) or \text{'n'}, this array contains the local pieces of the distributed matrix \(\text{sub}(A)\).

\(ia, ja\) (global) The row and column indices in the distributed matrix \(A\) indicating the first row and the first column of the submatrix \(\text{sub}(A)\), respectively.

desca (global and local) array of dimension 9. The array descriptor of the distributed matrix \(A\).

\(beta\) (global)
Specifies the scalar \(beta\).

c (local)
Array, size \((lld_c, \text{LOCq}(jc+n-1))\).

Before entry with \(\text{uplo} = \text{'U'}\) or \text{'u'}, this array contains \(n\)-by-\(n\) upper triangular part of the symmetric distributed matrix \(\text{sub}(C)\) and its strictly lower triangular part is not referenced.

Before entry with \(\text{uplo} = \text{'L'}\) or \text{'l'}, this array contains \(n\)-by-\(n\) lower triangular part of the symmetric distributed matrix \(\text{sub}(C)\) and its strictly upper triangular part is not referenced.

\(ic, jc\) (global) The row and column indices in the distributed matrix \(C\) indicating the first row and the first column of the submatrix \(\text{sub}(C)\), respectively.

descC (global and local) array of dimension 9. The array descriptor of the distributed matrix \(C\).

Output Parameters

c With \(\text{uplo} = \text{'U'}\) or \text{'u'}, the upper triangular part of \(\text{sub}(C)\) is overwritten by the upper triangular part of the updated distributed matrix.

With \(\text{uplo} = \text{'L'}\) or \text{'l'}, the lower triangular part of \(\text{sub}(C)\) is overwritten by the upper triangular part of the updated distributed matrix.

**p?her2k**

Performs a rank-2\(k\) update of a Hermitian distributed matrix.

**Syntax**

```c
void pcher2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const float *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descC );
```

```c
void pzher2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const double *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descC );
```
Include Files

- mkl_pblas.h

Description

The \texttt{p?her2k} routines perform a distributed matrix-matrix operation defined as

\[ \text{sub}(C):= \alpha \text{sub}(A) \ast \text{conjg}(\text{sub}(B))' + \text{conjg}(\alpha) \ast \text{sub}(B) \ast \text{conjg}(\text{sub}(A))' + \beta \ast \text{sub}(C), \]

or

\[ \text{sub}(C):= \alpha \ast \text{conjg}(\text{sub}(A))' \ast \text{sub}(A) + \text{conjg}(\alpha) \ast \text{conjg}(\text{sub}(B))' \ast \text{sub}(A) + \beta \ast \text{sub}(C), \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(C) \) is an \( n \)-by-\( n \) Hermitian distributed matrix, \( \text{sub}(C) = C(\text{ic}:\text{ic}+n-1, \text{jc}:\text{jc}+n-1) \).
- \( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+k-1) \), if \( \text{trans} = 'N' \) or \( 'n' \), and \( \text{sub}(A) = A(\text{ia}:\text{ia}+k-1, \text{ja}:\text{ja}+n-1) \) otherwise.
- \( \text{sub}(B) \) is a distributed matrix, \( \text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+k-1) \), if \( \text{trans} = 'N' \) or \( 'n' \), and \( \text{sub}(B) = B(\text{ib}:\text{ib}+k-1, \text{jb}:\text{jb}+n-1) \) otherwise.

Input Parameters

- **uplo** (global) Specifies whether the upper or lower triangular part of the Hermitian distributed matrix \( \text{sub}(C) \) is used:
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(C) \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(C) \) is used.

- **trans** (global) Specifies the operation:
  - If \( \text{trans} = 'N' \) or \( 'n' \), then \( \text{sub}(C) := \alpha \ast \text{sub}(A) \ast \text{conjg}(\text{sub}(B))' + \text{conjg}(\alpha) \ast \text{sub}(B) \ast \text{conjg}(\text{sub}(A))' + \beta \ast \text{sub}(C) \);   
  - If \( \text{trans} = 'C' \) or \( 'c' \), then \( \text{sub}(C) := \alpha \ast \text{conjg}(\text{sub}(A))' \ast \text{sub}(A) + \text{conjg}(\alpha) \ast \text{conjg}(\text{sub}(B))' \ast \text{sub}(A) + \beta \ast \text{sub}(C) \).

- **n** (global) Specifies the order of the distributed matrix \( \text{sub}(C) \), \( n \geq 0 \).

- **k** (global) On entry with \( \text{trans} = 'N' \) or \( 'n' \), \( k \) specifies the number of columns of the distributed matrices \( \text{sub}(A) \) and \( \text{sub}(B) \), and on entry with \( \text{trans} = 'C' \) or \( 'c' \), \( k \) specifies the number of rows of the distributed matrices \( \text{sub}(A) \) and \( \text{sub}(B) \), \( k \geq 0 \).

- **alpha** (global)
  - Specifies the scalar \( \alpha \).

- **a** (local)
  - Array, size \( (\text{lld}_a, \text{kla}) \), where \( \text{kla} \) is \( \text{LOCq}(\text{ja}+k-1) \) when \( \text{trans} = 'N' \) or \( 'n' \), and is \( \text{LOCq}(\text{ja}+n-1) \) otherwise. Before entry with \( \text{trans} = 'N' \) or \( 'n' \), this array contains the local pieces of the distributed matrix \( \text{sub}(A) \).

- **ia, ja** (global)
  - The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.
(global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

Array, size $(lld_{b}, klb)$, where $klb$ is $\text{LOCq}(jb+k-1)$ when $\text{trans} = 'N'$ or 'n', and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with $\text{trans} = 'N'$ or 'n', this array contains the local pieces of the distributed matrix $\text{sub}(B)$.

(global) The row and column indices in the distributed matrix $B$ indicating the first row and the first column of the submatrix $\text{sub}(B)$, respectively.

(global and local) array of dimension 9. The array descriptor of the distributed matrix $B$.

Specifies the scalar $\beta$.

Array, size $(lld_{c}, \text{LOCq}(jc+n-1))$.

Before entry with $\text{uplo} = 'U'$ or 'u', this array contains $n$-by-$n$ upper triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly lower triangular part is not referenced.

Before entry with $\text{uplo} = 'L'$ or 'l', this array contains $n$-by-$n$ lower triangular part of the symmetric distributed matrix $\text{sub}(C)$ and its strictly upper triangular part is not referenced.

(global) The row and column indices in the distributed matrix $C$ indicating the first row and the first column of the submatrix $\text{sub}(C)$, respectively.

(global and local) array of dimension 9. The array descriptor of the distributed matrix $C$.

With $\text{uplo} = 'U'$ or 'u', the upper triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

With $\text{uplo} = 'L'$ or 'l', the lower triangular part of $\text{sub}(C)$ is overwritten by the upper triangular part of the updated distributed matrix.

**pssymm**

*Performs a scalar-matrix-matrix product (one matrix operand is symmetric) and adds the result to a scalar-matrix product for distribute matrices.*

**Syntax**

```c
void pssymm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desc_a, const float *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *desc_b, const float *beta, float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *desc_c);
```
void pdsymm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const double *beta, double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descb);

void pcsymm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex8 *beta, const MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descb);

void pzsymm (const char *side, const char *uplo, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex16 *beta, const MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descb);

Include Files

• mkl_pblas.h

Description

The p?symm routines perform a matrix-matrix operation with distributed matrices. The operation is defined as

\[ \text{sub}(C) := \alpha \text{sub}(A) \text{sub}(B) + \beta \text{sub}(C), \]

or

\[ \text{sub}(C) := \alpha \text{sub}(B) \text{sub}(A) + \beta \text{sub}(C), \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(A) \) is a symmetric distributed matrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1), \) if \( \text{side} = 'L', \) and \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1), \) if \( \text{side} = 'R'. \)
- \( \text{sub}(B) \) and \( \text{sub}(C) \) are \( m \)-by-\( n \) distributed matrices.
- \( \text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1), \text{sub}(C) = C(ic:ic+m-1, jc:jc+n-1). \)

Input Parameters

**side**

(global) Specifies whether the symmetric distributed matrix \( \text{sub}(A) \) appears on the left or right in the operation:

- if \( \text{side} = 'L' \) or 'l', then \( \text{sub}(C) := \alpha \text{sub}(A) \text{sub}(B) + \beta \text{sub}(C); \)
- if \( \text{side} = 'R' \) or 'r', then \( \text{sub}(C) := \alpha \text{sub}(B) \text{sub}(A) + \beta \text{sub}(C). \)

**uplo**

(global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \( \text{sub}(A) \) is used:

- if \( \text{uplo} = 'U' \) or 'u', then the upper triangular part is used;
- if \( \text{uplo} = 'L' \) or 'l', then the lower triangular part is used.
\( m \) (global) Specifies the number of rows of the distribute submatrix \( \text{sub}(C) \), \( m \geq 0 \).

\( n \) (global) Specifies the number of columns of the distribute submatrix \( \text{sub}(C) \), \( n \geq 0 \).

\( alpha \) (global) Specifies the scalar \( alpha \).

\( a \) (local) Array, size \((\text{lld}_a, \text{LOCq}(ja+na-1))\).

Before entry this array must contain the local pieces of the symmetric distributed matrix \( \text{sub}(A) \), such that when \( \text{uplo} = 'U' \) or \( 'u' \), the \( na \)-by-\( na \) upper triangular part of the distributed matrix \( \text{sub}(A) \) must contain the upper triangular part of the symmetric distributed matrix and the strictly lower triangular part of \( \text{sub}(A) \) is not referenced, and when \( \text{uplo} = 'L' \) or \( 'l' \), the \( na \)-by-\( na \) lower triangular part of the distributed matrix \( \text{sub}(A) \) must contain the lower triangular part of the symmetric distributed matrix and the strictly upper triangular part of \( \text{sub}(A) \) is not referenced.

\( ia, ja \) (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

\( desca \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

\( b \) (local) Array, size \((\text{lld}_b, \text{LOCq}(jb+n-1))\). Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(B) \).

\( ib, jb \) (global) The row and column indices in the distributed matrix \( B \) indicating the first row and the first column of the submatrix \( \text{sub}(B) \), respectively.

\( descb \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( B \).

\( beta \) (global) Specifies the scalar \( beta \).

When \( beta \) is set to zero, then \( \text{sub}(C) \) need not be set on input.

\( c \) (local) Array, size \((\text{lld}_c, \text{LOCq}(jc+n-1))\). Before entry this array must contain the local pieces of the distributed matrix \( \text{sub}(C) \).

\( ic, jc \) (global) The row and column indices in the distributed matrix \( C \) indicating the first row and the first column of the submatrix \( \text{sub}(C) \), respectively.

\( desc\text{c} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( C \).

**Output Parameters**

\( c \) Overwritten by the \( m \)-by-\( n \) updated matrix.
p?syrk

Performs a rank-k update of a symmetric distributed matrix.

Syntax

```c
void pssyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const float *alpha , const float *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const float *beta , float *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
void pdsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const double *alpha , const double *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const double *beta , double *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
void pcsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const MKL_Complex8 *alpha , const MKL_Complex8 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex8 *beta , MKL_Complex8 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
void pzsyrk (const char *uplo , const char *trans , const MKL_INT *n , const MKL_INT *k , const MKL_Complex16 *alpha , const MKL_Complex16 *a , const MKL_INT *ia , const MKL_INT *ja , const MKL_INT *desca , const MKL_Complex16 *beta , MKL_Complex16 *c , const MKL_INT *ic , const MKL_INT *jc , const MKL_INT *descc );
```

Include Files

- mkl_pblas.h

Description

The p?syrk routines perform a distributed matrix-matrix operation defined as

\[ \text{sub}(C) := \alpha \text{sub}(A) \text{sub}(A) + \beta \text{sub}(C), \]

or

\[ \text{sub}(C) := \alpha \text{sub}(A) \text{sub}(A) + \beta \text{sub}(C), \]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(C) \) is an \( n \)-by-\( n \) symmetric distributed matrix, \( \text{sub}(C) = C(\text{ic:ic+n-1, jc:jc+n-1}) \).
- \( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia:ia+n-1, ja:ja+k-1}) \), if \( \text{trans} = 'N' \) or \( 'n' \), and \( \text{sub}(A) = A(\text{ia:ia+k-1, ja:ja+n-1}) \) otherwise.

Input Parameters

- **uplo** (global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \( \text{sub}(C) \) is used:
  - If \( \text{uplo} = 'U' \) or \( 'u' \), then the upper triangular part of the \( \text{sub}(C) \) is used.
  - If \( \text{uplo} = 'L' \) or \( 'l' \), then the lower triangular part of the \( \text{sub}(C) \) is used.

- **trans** (global) Specifies the operation:
if \textit{trans} = 'N' or 'n', then \( \text{sub}(C) := \alpha \cdot \text{sub}(A) \cdot \text{sub}(A)' + \beta \cdot \text{sub}(C); \)

if \textit{trans} = 'T' or 't', then \( \text{sub}(C) := \alpha \cdot \text{sub}(A)' \cdot \text{sub}(A) + \beta \cdot \text{sub}(C). \)

\( n \) (global) Specifies the order of the distributed matrix \( \text{sub}(C) \), \( n \geq 0 \).

\( k \) (global) On entry with \textit{trans} = 'N' or 'n', \( k \) specifies the number of columns of the distributed matrix \( \text{sub}(A) \), and on entry with \textit{trans} = 'T' or 't', \( k \) specifies the number of rows of the distributed matrix \( \text{sub}(A) \), \( k \geq 0 \).

\( \alpha \) (global)

Specifies the scalar \( \alpha \).

\( \alpha \) (local)

Array, size \((\text{lld}_a, \text{kla})\), where \( \text{kla} \) is \( \text{LOCq}(ja+k-1) \) when \textit{trans} = 'N' or 'n', and is \( \text{LOCq}(ja+n-1) \) otherwise. Before entry with \textit{trans} = 'N' or 'n', this array contains the local pieces of the distributed matrix \( \text{sub}(A) \).

\( \text{ia}, \text{ja} \) (global) The row and column indices in the distributed matrix \( \text{A} \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( \text{A} \).

\( \beta \) (global)

Specifies the scalar \( \beta \).

\( \beta \) (local)

Array, size \((\text{lld}_c, \text{LOCq}(jc+n-1))\).

Before entry with \textit{uplo} = 'U' or 'u', this array contains \( n \)-by-\( n \) upper triangular part of the symmetric distributed matrix \( \text{sub}(C) \) and its strictly lower triangular part is not referenced.

Before entry with \textit{uplo} = 'L' or 'l', this array contains \( n \)-by-\( n \) lower triangular part of the symmetric distributed matrix \( \text{sub}(C) \) and its strictly upper triangular part is not referenced.

\( \text{ic}, \text{jc} \) (global) The row and column indices in the distributed matrix \( \text{C} \) indicating the first row and the first column of the submatrix \( \text{sub}(C) \), respectively.

\( \text{descC} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( \text{C} \).

\textbf{Output Parameters}

\( \text{c} \) With \textit{uplo} = 'U' or 'u', the upper triangular part of \( \text{sub}(C) \) is overwritten by the upper triangular part of the updated distributed matrix.

With \textit{uplo} = 'L' or 'l', the lower triangular part of \( \text{sub}(C) \) is overwritten by the upper triangular part of the updated distributed matrix.
P?SYR2K

Performs a rank-2k update of a symmetric distributed matrix.

Syntax

```c
void pssyr2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const float *beta, float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pdsyr2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const double *beta, double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pcsyr2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex8 *beta, const MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);

void pzsyr2k (const char *uplo, const char *trans, const MKL_INT *n, const MKL_INT *k, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb, const MKL_Complex16 *beta, const MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```

Include Files

- mkl_pblas.h

Description

The P?SYR2K routines perform a distributed matrix-matrix operation defined as

\[
\text{sub}(C) := \alpha \text{sub}(A) \ast \text{sub}(B) + \text{sub}(B) \ast \text{sub}(A) + \beta \text{sub}(C),
\]

or

\[
\text{sub}(C) := \alpha \text{sub}(A) \ast \text{sub}(B) + \alpha \text{sub}(B) \ast \text{sub}(A) + \beta \text{sub}(C),
\]

where:

- \(\alpha\) and \(\beta\) are scalars,
- \(\text{sub}(C)\) is an \(n\)-by-\(n\) symmetric distributed matrix, \(\text{sub}(C) = C(\text{ic}:\text{ic}+n-1, \text{jc}:\text{jc}+n-1)\).
- \(\text{sub}(A)\) is a distributed matrix, \(\text{sub}(A) = A(\text{ia}:\text{ia}+n-1, \text{ja}:\text{ja}+k-1)\), if \(\text{trans} = 'N'\) or 'n', and \(\text{sub}(A) = A(\text{ia}:\text{ia}+k-1, \text{ja}:\text{ja}+n-1)\) otherwise.
- \(\text{sub}(B)\) is a distributed matrix, \(\text{sub}(B) = B(\text{ib}:\text{ib}+n-1, \text{jb}:\text{jb}+k-1)\), if \(\text{trans} = 'N'\) or 'n', and \(\text{sub}(B) = B(\text{ib}:\text{ib}+k-1, \text{jb}:\text{jb}+n-1)\) otherwise.

Input Parameters

- `uplo` (global) Specifies whether the upper or lower triangular part of the symmetric distributed matrix \(\text{sub}(C)\) is used:
If $uplo = 'U' \text{ or } 'u'$, then the upper triangular part of the $sub(C)$ is used.

If $uplo = 'L' \text{ or } 'l'$, then the low triangular part of the $sub(C)$ is used.

$trans$ (global) Specifies the operation:

if $trans = 'N' \text{ or } 'n'$, then $sub(C) := alpha*sub(A)*sub(B)' + alpha*sub(B)*sub(A)' + beta*sub(C);

if $trans = 'T' \text{ or } 't'$, then $sub(C) := alpha*sub(B)'*sub(A) + alpha*sub(A)'*sub(B) + beta*sub(C)$.

$n$ (global) Specifies the order of the distributed matrix $sub(C)$, $n \geq 0$.

$k$ (global) On entry with $trans = 'N' \text{ or } 'n'$, $k$ specifies the number of columns of the distributed matrices $sub(A)$ and $sub(B)$, and on entry with $trans = 'T' \text{ or } 't'$, $k$ specifies the number of rows of the distributed matrices $sub(A)$ and $sub(B)$, $k \geq 0$.

$alpha$ (global)

Specifies the scalar $alpha$.

$a$ (local)

Array, size $(lld_a, kla)$, where $kla$ is $\text{LOCq}(ja+k-1)$ when $trans = 'N$ or 'n', and is $\text{LOCq}(ja+n-1)$ otherwise. Before entry with $trans = 'N$ or 'n', this array contains the local pieces of the distributed matrix $sub(A)$.

$ia, ja$ (global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix $sub(A)$, respectively.

$desca$ (global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

$b$ (local)

Array, size $(lld_b, klb)$, where $klb$ is $\text{LOCq}(jb+k-1)$ when $trans = 'N' \text{ or } 'n'$, and is $\text{LOCq}(jb+n-1)$ otherwise. Before entry with $trans = 'N' \text{ or } 'n'$, this array contains the local pieces of the distributed matrix $sub(B)$.

$ib, jb$ (global) The row and column indices in the distributed matrix $B$ indicating the first row and the first column of the submatrix $sub(B)$, respectively.

$descb$ (global and local) array of dimension 9. The array descriptor of the distributed matrix $B$.

$beta$ (global)

Specifies the scalar $beta$.

$c$ (local)

Array, size $(lld_c, \text{LOCq}(jc+n-1))$.

Before entry with $uplo = 'U' \text{ or } 'u'$, this array contains $n$-by-$n$ upper triangular part of the symmetric distributed matrix $sub(C)$ and its strictly lower triangular part is not referenced.
Before entry with \( \text{uplo} = 'L' \) or \( 'l' \), this array contains \( n \)-by-\( n \) lower triangular part of the symmetric distributed matrix \( \text{sub}(C) \) and its strictly upper triangular part is not referenced.

\( ic, jc \) (global) The row and column indices in the distributed matrix \( C \) indicating the first row and the first column of the submatrix \( \text{sub}(C) \), respectively.

\( \text{desc} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( C \).

**Output Parameters**

\( c \) With \( \text{uplo} = 'U' \) or \( 'u' \), the upper triangular part of \( \text{sub}(C) \) is overwritten by the upper triangular part of the updated distributed matrix.

With \( \text{uplo} = 'L' \) or \( 'l' \), the lower triangular part of \( \text{sub}(C) \) is overwritten by the upper triangular part of the updated distributed matrix.

\textbf{\( \text{p?tran} \)}

*Transposes a real distributed matrix.*

**Syntax**

```c
void pstran (const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const float *beta, float *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *desc );
void pdtran (const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const double *beta, double *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *desc );
```

**Include Files**

- `mkl_pblas.h`

**Description**

The \( \text{p?tran} \) routines transpose a real distributed matrix. The operation is defined as

\[
\text{sub}(C) := \beta \cdot \text{sub}(C) + \alpha \cdot \text{sub}(A)',
\]

where:

- \( \alpha \) and \( \beta \) are scalars,
- \( \text{sub}(C) \) is an \( m \)-by-\( n \) distributed matrix, \( \text{sub}(C) = C(\text{ic} : \text{ic} + m - 1, \text{jc} : \text{jc} + n - 1) \).
- \( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia} : \text{ia} + n - 1, \text{ja} : \text{ja} + m - 1) \).

**Input Parameters**

\( m \) (global) Specifies the number of rows of the distributed matrix \( \text{sub}(C) \), \( m \geq 0 \).

\( n \) (global) Specifies the number of columns of the distributed matrix \( \text{sub}(C) \), \( n \geq 0 \).

\( \alpha \) (global)
Specifies the scalar $\alpha$.

$a$

(local) Array, size $(l1d_a, LOCq(ja+m-1))$. This array contains the local pieces of the distributed matrix sub($A$).

$ia, ja$

(global) The row and column indices in the distributed matrix $A$ indicating the first row and the first column of the submatrix sub($A$), respectively.

$desca$

(global and local) array of dimension 9. The array descriptor of the distributed matrix $A$.

$\beta$

(global) Specifies the scalar $\beta$.

When $\beta$ is equal to zero, then sub($C$) need not be set on input.

$c$

(local) Array, size $(l1d_c, LOCq(jc+n-1))$. This array contains the local pieces of the distributed matrix sub($C$).

$ic, jc$

(global) The row and column indices in the distributed matrix $C$ indicating the first row and the first column of the submatrix sub($C$), respectively.

$descc$

(global and local) array of dimension 9. The array descriptor of the distributed matrix $C$.

Output Parameters

c

Overwritten by the updated submatrix.

$p\text{tranu}$

Transposes a distributed complex matrix.

Syntax

```c
void pctransu (const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
void pztranu (const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```

Include Files

• mkl_pblas.h

Description

The $p\text{tranu}$ routines transpose a complex distributed matrix. The operation is defined as

$$\text{sub}(C):=\beta\text{sub}(C) + \alpha\text{sub}(A)' ,$$

where:
alpha and beta are scalars, 

\( \text{sub}(C) \) is an \( m \)-by-\( n \) distributed matrix, \( \text{sub}(C) = C(\text{ic:ic}+m-1, \text{jc:jc}+n-1) \). 

\( \text{sub}(A) \) is a distributed matrix, \( \text{sub}(A) = A(\text{ia:ia}+n-1, \text{ja:ja}+m-1) \).

Input Parameters

\( m \) (global) Specifies the number of rows of the distributed matrix \( \text{sub}(C) \), \( m \geq 0 \).

\( n \) (global) Specifies the number of columns of the distributed matrix \( \text{sub}(C) \), \( n \geq 0 \).

\( \alpha \) (global) Specifies the scalar \( \alpha \).

\( a \) (local) Array, size \((\text{lld}_a, \text{LOCq}(\text{ja}+m-1))\). This array contains the local pieces of the distributed matrix \( \text{sub}(A) \).

\( \text{ia}, \text{ja} \) (global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

\( \text{desca} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

\( \beta \) (global) Specifies the scalar \( \beta \).

When \( \beta \) is equal to zero, then \( \text{sub}(C) \) need not be set on input.

\( c \) (local) Array, size \((\text{lld}_c, \text{LOCq}(\text{jc}+n-1))\). This array contains the local pieces of the distributed matrix \( \text{sub}(C) \).

\( \text{ic}, \text{jc} \) (global) The row and column indices in the distributed matrix \( C \) indicating the first row and the first column of the submatrix \( \text{sub}(C) \), respectively.

\( \text{descc} \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( C \).

Output Parameters

\( c \) Overwritten by the updated submatrix.

\( \text{p?tranc} \)

Transposes a complex distributed matrix, conjugated.

Syntax

```c
void pctranc (const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex8 *beta, MKL_Complex8 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descc);
```
void pztranc (const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, const MKL_Complex16 *beta, MKL_Complex16 *c, const MKL_INT *ic, const MKL_INT *jc, const MKL_INT *descC);

**Include Files**
- mkl_pblas.h

**Description**
The `p?tranc` routines transpose a complex distributed matrix. The operation is defined as

\[
\text{sub}(\mathbf{C}) := \beta \cdot \text{sub}(\mathbf{C}) + \alpha \cdot \text{conjg}(\text{sub}(\mathbf{A})),
\]

where:
- *alpha* and *beta* are scalars,
- *sub*(\(\mathbf{C}\)) is an \(m\)-by-\(n\) distributed matrix, \(\text{sub}(\mathbf{C})=\mathbf{C}(\text{ic:ic}+m-1, \text{jc:jc}+n-1)\).
- *sub*(\(\mathbf{A}\)) is a distributed matrix, \(\text{sub}(\mathbf{A})=\mathbf{A}(\text{ia:ia}+n-1, \text{ja:ja}+m-1)\).

**Input Parameters**

- **m** (global) Specifies the number of rows of the distributed matrix \(\text{sub}(\mathbf{C})\), \(m \geq 0\).
- **n** (global) Specifies the number of columns of the distributed matrix \(\text{sub}(\mathbf{C})\), \(n \geq 0\).
- **alpha** (global) Specifies the scalar *alpha*.
- **a** (local) Array, size \((\text{lld}_a, \text{LOCq}(\text{ja}+m-1))\). This array contains the local pieces of the distributed matrix \(\text{sub}(\mathbf{A})\).
- **ia, ja** (global) The row and column indices in the distributed matrix \(\mathbf{A}\) indicating the first row and the first column of the submatrix \(\text{sub}(\mathbf{A})\), respectively.
- **desca** (global and local) array of dimension 9. The array descriptor of the distributed matrix \(\mathbf{A}\).
- **beta** (global) Specifies the scalar *beta*.
  
  When *beta* is equal to zero, then \(\text{sub}(\mathbf{C})\) need not be set on input.
- **c** (local) Array, size \((\text{lld}_c, \text{LOCq}(\text{jc}+n-1))\). This array contains the local pieces of the distributed matrix \(\text{sub}(\mathbf{C})\).
- **ic, jc** (global) The row and column indices in the distributed matrix \(\mathbf{C}\) indicating the first row and the first column of the submatrix \(\text{sub}(\mathbf{C})\), respectively.
- **descC** (global and local) array of dimension 9. The array descriptor of the distributed matrix \(\mathbf{C}\).
Output Parameters

c
Overwritten by the updated submatrix.

**p?trmm**
Computes a scalar-matrix-matrix product (one matrix operand is triangular) for distributed matrices.

**Syntax**

```c
void pstrmm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, float *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb );
void pdtrmm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, double *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb );
void pctrmm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb );
void pztrmm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb );
```

**Include Files**

- mkl_pblas.h

**Description**

The `p?trmm` routines perform a matrix-matrix operation using triangular matrices. The operation is defined as

\[
\text{sub}(B) := \alpha \cdot \text{op}(\text{sub}(A)) \cdot \text{sub}(B)
\]

or

\[
\text{sub}(B) := \alpha \cdot \text{sub}(B) \cdot \text{op}(\text{sub}(A))
\]

where:

- \(\alpha\) is a scalar,
- \(\text{sub}(B)\) is an \(m\times n\) distributed matrix, \(\text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1)\).
- \(A\) is a unit, or non-unit, upper or lower triangular distributed matrix, \(\text{sub}(A) = A(ia:ia+m-1, ja:ja+n-1)\), if \(\text{side} = 'L'\) or 'l', and \(\text{sub}(A) = A(ia:ia+n-1, ja:ja+m-1)\), if \(\text{side} = 'R'\) or 'r'.
- \(\text{op}(\text{sub}(A))\) is one of \(\text{op}(\text{sub}(A)) = \text{sub}(A)\), or \(\text{op}(\text{sub}(A)) = \text{sub}(A)'\), or \(\text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A))'\).

**Input Parameters**

- **side**
  (global) Specifies whether \(\text{op}(\text{sub}(A))\) appears on the left or right of \(\text{sub}(B)\) in the operation:
if `side = 'L'` or `l'`, then \( \text{sub}(B) := \alpha \cdot \text{op}(\text{sub}(A)) \cdot \text{sub}(B) \);
if `side = 'R'` or `r'`, then \( \text{sub}(B) := \alpha \cdot \text{sub}(B) \cdot \text{op}(\text{sub}(A)) \).

**uplo**

(global) Specifies whether the distributed matrix \( \text{sub}(A) \) is upper or lower triangular:
if `uplo = 'U'` or `u'`, then the matrix is upper triangular;
if `uplo = 'L'` or `l'`, then the matrix is lower triangular.

**transa**

(global) Specifies the form of \( \text{op}(\text{sub}(A)) \) used in the matrix multiplication:
if `transa = 'N'` or `n'`, then \( \text{op}(\text{sub}(A)) = \text{sub}(A) \);
if `transa = 'T'` or `t'`, then \( \text{op}(\text{sub}(A)) = \text{sub}(A)' \);
if `transa = 'C'` or `c'`, then \( \text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)') \).

**diag**

(global) Specifies whether the matrix \( \text{sub}(A) \) is unit triangular:
if `diag = 'U'` or `u'`, then the matrix is unit triangular;
if `diag = 'N'` or `n'`, then the matrix is not unit triangular.

**m**

(global) Specifies the number of rows of the distributed matrix \( \text{sub}(B) \), \( m \geq 0 \).

**n**

(global) Specifies the number of columns of the distributed matrix \( \text{sub}(B) \), \( n \geq 0 \).

**alpha**

(global)
Specifies the scalar \( \alpha \).
When \( \alpha \) is zero, then the array \( b \) need not be set before entry.

**a**

(local)
Array, size \( lld_a \) by \( ka \), where \( ka \) is at least \( \text{LOCq}(1, ja+m-1) \) when `side = 'L'` or `l'` and is at least \( \text{LOCq}(1, ja+n-1) \) when `side = 'R'` or `r'`.
Before entry with `uplo = 'U'` or `u'`, this array contains the local entries corresponding to the entries of the upper triangular distributed matrix \( \text{sub}(A) \), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix \( \text{sub}(A) \) is not referenced.
Before entry with `uplo = 'L'` or `l'`, this array contains the local entries corresponding to the entries of the lower triangular distributed matrix \( \text{sub}(A) \), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix \( \text{sub}(A) \) is not referenced.
When `diag = 'U'` or `u'`, the local entries corresponding to the diagonal elements of the submatrix \( \text{sub}(A) \) are not referenced either, but are assumed to be unity.

**ia, ja**

(global) The row and column indices in the distributed matrix \( A \) indicating the first row and the first column of the submatrix \( \text{sub}(A) \), respectively.

**desca**

(global and local) array of dimension 9. The array descriptor of the distributed matrix \( A \).

**b**

(local)
Array, size (lld_b, LOCq(1, jb+n-1)).

Before entry, this array contains the local pieces of the distributed matrix sub(B).

\( ib, jb \) (global) The row and column indices in the distributed matrix \( B \) indicating the first row and the first column of the submatrix \( \text{sub}(B) \), respectively.

\( descb \) (global and local) array of dimension 9. The array descriptor of the distributed matrix \( B \).

Output Parameters

\( b \) Overwritten by the transformed distributed matrix.

\texttt{p?trsm}

Solves a distributed matrix equation (one matrix operand is triangular).

Syntax

void pstrsm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const float *alpha, const float *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, float *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb);

void pdtrsm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const double *alpha, const double *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, double *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb);

void pctrsm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const MKL_Complex8 *alpha, const MKL_Complex8 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex8 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb);

void pztrsm (const char *side, const char *uplo, const char *transa, const char *diag, const MKL_INT *m, const MKL_INT *n, const MKL_Complex16 *alpha, const MKL_Complex16 *a, const MKL_INT *ia, const MKL_INT *ja, const MKL_INT *desca, MKL_Complex16 *b, const MKL_INT *ib, const MKL_INT *jb, const MKL_INT *descb);

Include Files

- mkl_pblas.h

Description

The \texttt{p?trsm} routines solve one of the following distributed matrix equations:

\[ \text{op}(\text{sub}(A)) \times X = \alpha \times \text{sub}(B), \]

or

\[ X \times \text{op}(\text{sub}(A)) = \alpha \times \text{sub}(B), \]

where:

\( \alpha \) is a scalar,

\( X \) and \( \text{sub}(B) \) are \( m \)-by-\( n \) distributed matrices, \( \text{sub}(B) = B(ib:ib+m-1, jb:jb+n-1) \);
A is a unit, or non-unit, upper or lower triangular distributed matrix, \( \text{sub}(A) = A(ia:ia+m-1, ja:ja+m-1) \), if \( \text{side} = 'L' \) or \( 'l' \), and \( \text{sub}(A) = A(ia:ia+n-1, ja:ja+n-1) \), if \( \text{side} = 'R' \) or \( 'r' \);

\( \text{op}(\text{sub}(A)) \) is one of \( \text{op}(\text{sub}(A)) = \text{sub}(A) \), or \( \text{op}(\text{sub}(A)) = \text{sub}(A)' \), or \( \text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)') \).

The distributed matrix \( \text{sub}(B) \) is overwritten by the solution matrix \( X \).

**Input Parameters**

- **side** (global) Specifies whether \( \text{op}(\text{sub}(A)) \) appears on the left or right of \( X \) in the equation:
  - if \( \text{side} = 'L' \) or \( 'l' \), then \( \text{op}(\text{sub}(A)) \cdot X = \alpha \cdot \text{sub}(B) \);
  - if \( \text{side} = 'R' \) or \( 'r' \), then \( X \cdot \text{op}(\text{sub}(A)) = \alpha \cdot \text{sub}(B) \).

- **uplo** (global) Specifies whether the distributed matrix \( \text{sub}(A) \) is upper or lower triangular:
  - if \( \text{uplo} = 'U' \) or \( 'u' \), then the matrix is upper triangular;
  - if \( \text{uplo} = 'L' \) or \( 'l' \), then the matrix is lower triangular.

- **transa** (global) Specifies the form of \( \text{op}(\text{sub}(A)) \) used in the matrix equation:
  - if \( \text{transa} = 'N' \) or \( 'n' \), then \( \text{op}(\text{sub}(A)) = \text{sub}(A) \);
  - if \( \text{transa} = 'T' \) or \( 't' \), then \( \text{op}(\text{sub}(A)) = \text{sub}(A)' \);
  - if \( \text{transa} = 'C' \) or \( 'c' \), then \( \text{op}(\text{sub}(A)) = \text{conjg}(\text{sub}(A)') \).

- **diag** (global) Specifies whether the matrix \( \text{sub}(A) \) is unit triangular:
  - if \( \text{diag} = 'U' \) or \( 'u' \) then the matrix is unit triangular;
  - if \( \text{diag} = 'N' \) or \( 'n' \), then the matrix is not unit triangular.

- **m** (global) Specifies the number of rows of the distributed matrix \( \text{sub}(B) \), \( m \geq 0 \).

- **n** (global) Specifies the number of columns of the distributed matrix \( \text{sub}(B) \), \( n \geq 0 \).

- **alpha** (global)
  Specifies the scalar \( \alpha \).
  When \( \alpha \) is zero, then \( A \) is not referenced and \( B \) need not be set before entry.

- **a** (local)
  Array, size \( \text{ld}_a \times \text{ka} \), where \( \text{ka} \) is at least \( \text{loc}(1, ja+m-1) \) when \( \text{side} = 'L' \) or \( 'l' \) and is at least \( \text{loc}(1, ja+n-1) \) when \( \text{side} = 'R' \) or \( 'r' \).
  Before entry with \( \text{uplo} = 'U' \) or \( 'u' \), this array contains the local entries corresponding to the entries of the upper triangular distributed matrix \( \text{sub}(A) \), and the local entries corresponding to the entries of the strictly lower triangular part of the distributed matrix \( \text{sub}(A) \) is not referenced.
Before entry with uplo = 'L' or 'l', this array contains the local entries corresponding to the entries of the lower triangular distributed matrix sub(A), and the local entries corresponding to the entries of the strictly upper triangular part of the distributed matrix sub(A) is not referenced.

When diag = 'U' or 'u', the local entries corresponding to the diagonal elements of the submatrix sub(A) are not referenced either, but are assumed to be unity.

\( ia, ja \) (global) The row and column indices in the distributed matrix A indicating the first row and the first column of the submatrix sub(A), respectively.

\( desca \) (global and local) array of dimension 9. The array descriptor of the distributed matrix A.

\( b \) (local)

Array, size (lld_b, LOCq(1, jb+n-1)).

Before entry, this array contains the local pieces of the distributed matrix sub(B).

\( ib, jb \) (global) The row and column indices in the distributed matrix B indicating the first row and the first column of the submatrix sub(B), respectively.

\( descb \) (global and local) array of dimension 9. The array descriptor of the distributed matrix B.

**Output Parameters**

\( b \) Overwritten by the solution distributed matrix X.
The Intel® Math Kernel Library (Intel® MKL) provides tools for solving Partial Differential Equations (PDE). These tools are Trigonometric Transform interface routines (see Trigonometric Transform Routines) and Poisson Solver (see Fast Poisson Solver Routines).

Poisson Solver is designed for fast solving of simple Helmholtz, Poisson, and Laplace problems. The solver is based on the Trigonometric Transform interface, which is, in turn, based on the Intel® MKL Fast Fourier Transform (FFT) interface (refer to Fourier Transform Functions), optimized for Intel® processors.

Direct use of the Trigonometric Transform routines may be helpful to those who have already implemented their own solvers similar to the Intel® MKL Poisson Solver. As it may be hard enough to modify the original code so as to make it work with Poisson Solver, you are encouraged to use fast (staggered) sine/cosine transforms implemented in the Trigonometric Transform interface to improve performance of your solver.

Both Trigonometric Transform and Poisson Solver routines can be called from C and Fortran.

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
<tr>
<td>Notice revision #20110804</td>
</tr>
</tbody>
</table>

Trigonometric Transform Routines

In addition to the Fast Fourier Transform (FFT) interface, described in chapter "Fast Fourier Transforms", Intel® MKL supports the Real Discrete Trigonometric Transforms (sometimes called real-to-real Discrete Fourier Transforms) interface. In this document, the interface is referred to as TT interface. It implements a group of routines (TT routines) used to compute sine/cosine, staggered sine/cosine, and twice staggered sine/cosine transforms (referred to as staggered2 sine/cosine transforms, for brevity). The TT interface provides much flexibility of use: you can adjust routines to your particular needs at the cost of manually tuning routine parameters or just call routines with default parameter values. The current Intel® MKL implementation of the TT interface can be used in solving partial differential equations and contains routines that are helpful for Fast Poisson and similar solvers.

For the list of Trigonometric Transforms currently implemented in Intel® MKL TT interface, see Transforms Implemented.

If you have got used to the FFTW interface (www.fftw.org), you can call the TT interface functions through real-to-real FFTW to Intel® MKL wrappers without changing FFTW function calls in your code (refer to FFTW to Intel® MKL Wrappers for FFTW 3.x for details). However, you are strongly encouraged to use the native TT interface for better performance. Another reason why you should use the wrappers cautiously is that TT and the real-to-real FFTW interfaces are not fully compatible and some features of the real-to-real FFTW, such as strides and multidimensional transforms, are not available through wrappers.

Trigonometric Transforms Implemented

TT routines allow computing the following transforms:
Forward sine transform

\[ F(k) = \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \frac{ki\pi}{n}, \quad k = 1, ..., n - 1 \]

Backward sine transform

\[ f(i) = \sum_{k=1}^{n-1} F(k) \sin \frac{ki\pi}{n}, \quad i = 1, ..., n - 1 \]

Forward staggered sine transform

\[ F(k) = \frac{1}{n} \sin \left(\frac{2k-1}{2}\right)\pi f(n) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \sin \left(\frac{(2k-1)i\pi}{2n}\right), \quad k = 1, ..., n \]

Backward staggered sine transform

\[ f(i) = \sum_{k=1}^{n} F(k) \sin \left(\frac{(2k-1)i\pi}{2n}\right), \quad i = 1, ..., n \]

Forward staggered2 sine transform

\[ F(k) = \frac{2}{n} \sum_{i=1}^{n} f(i) \sin \left(\frac{(2k-1)(2i-1)\pi}{4n}\right), \quad k = 1, ..., n \]

Backward staggered2 sine transform

\[ f(i) = \sum_{k=1}^{n} F(k) \sin \left(\frac{(2k-1)(2i-1)\pi}{4n}\right), \quad i = 1, ..., n \]

Forward cosine transform

\[ F(k) = \frac{1}{n} [f(0) + f(n)\cos k\pi] + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{ki\pi}{n}, \quad k = 0, ..., n \]

Backward cosine transform

\[ f(i) = \frac{1}{2} [F(0) + F(n)\cos i\pi] + \sum_{k=1}^{n-1} F(k) \cos \frac{ki\pi}{n}, \quad i = 0, ..., n \]

Forward staggered cosine transform

\[ F(k) = \frac{1}{n} f(0) + \frac{2}{n} \sum_{i=1}^{n-1} f(i) \cos \frac{(2k+1)i\pi}{2n}, \quad k = 0, ..., n - 1 \]

Backward staggered cosine transform

\[ f(i) = \sum_{k=0}^{n-1} F(k) \cos \frac{(2k+1)i\pi}{2n}, \quad i = 0, ..., n - 1 \]

Forward staggered2 cosine transform

\[ F(k) = \frac{2}{n} \sum_{i=1}^{n} f(i) \cos \left(\frac{(2k-1)(2i-1)\pi}{4n}\right), \quad k = 1, ..., n \]

Backward staggered2 cosine transform

\[ f(i) = \sum_{k=1}^{n} F(k) \cos \left(\frac{(2k-1)(2i-1)\pi}{4n}\right), \quad i = 1, ..., n \]
The size of the transform $n$ can be any integer greater or equal to 2.

Sequence of Invoking TT Routines

Computation of a transform using TT interface is conceptually divided into four steps, each of which is performed via a dedicated routine. Table "TT Interface Routines" lists the routines and briefly describes their purpose and use.

Most TT routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names.

## TT Interface Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?_init_trig_transform</td>
<td>Initializes basic data structures of Trigonometric Transforms.</td>
</tr>
<tr>
<td>?_commit_trig_transform</td>
<td>Checks consistency and correctness of user-defined data and creates a data structure to be used by Intel® MKL FFT interface.</td>
</tr>
<tr>
<td>?_forward_trig_transform</td>
<td>Computes a forward/backward Trigonometric Transform of a specified type using the appropriate formula (see Transforms Implemented).</td>
</tr>
<tr>
<td>?_backward_trig_transform</td>
<td></td>
</tr>
<tr>
<td>free_trig_transform</td>
<td>Releases the memory used by a data structure needed for calling FFT interface.</td>
</tr>
</tbody>
</table>

1 TT routines call Intel® MKL FFT interface for better performance.

To find a transformed vector for a particular input vector only once, the Intel® MKL TT interface routines are normally invoked in the order in which they are listed in Table "TT Interface Routines".

NOTE
Though the order of invoking TT routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure "Typical Order of Invoking TT Interface Routines" indicates the typical order in which TT interface routines can be invoked in a general case (prefixes and suffixes in routine names are omitted).
Typical Order of Invoking TT Interface Routines

```
Initialize
|<------------------------+
| | Change routine parameters (manually) |
| | Commit |
| |<------------------------+
| | forward/backward |
| |<------------------------+
| Free memory
```

A general scheme of using TT routines for double-precision computations is shown below. A similar scheme holds for single-precision computations with the only difference in the initial letter of routine names.

```c
...  
  d_init_trig_transform(&n, &tt_type, ipar, dpar, &ir);  
  /* Change parameters in ipar if necessary. */  
  /* Note that the result of the Transform will be in f. If you want to preserve the data stored in f,  
    save it to another location before the function call below */  
  d_commit_trig_transform(f, &handle, ipar, dpar, &ir);  
  d_forward_trig_transform(f, &handle, ipar, dpar, &ir);  
  d_backward_trig_transform(f, &handle, ipar, dpar, &ir);  
  free_trig_transform(&handle, ipar, &ir);  
  /* here the user may clean the memory used by f, dpar, ipar */  
...  
```

You can find examples of code that uses TT interface routines to solve one-dimensional Helmholtz problem in the examples\pdettc\source folder in your Intel® MKL directory.

Trigonometric Transform Interface Description

All types in this documentation are either standard C types float and double or MKL_INT integer type. For more information on the C types, refer to C Datatypes Specific to Intel® MKL and the Intel® MKL Developer Guide. To better understand usage of the types, see examples in the examples\pdettc\source folder in your Intel® MKL directory.

Routine Options

All TT routines use parameters to pass various options to one another. These parameters are arrays ipar, dpar and spar. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs.

**WARNING**

To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.
User Data Arrays

TT routines take arrays of user data as input. For example, user arrays are passed to the routine `d_forward_trig_transform` to compute a forward Trigonometric Transform. To minimize storage requirements and improve the overall run-time efficiency, Intel® MKL TT routines do not make copies of user input arrays.

NOTE
If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the Intel® MKL Developer Guide (search the document for coding techniques to improve performance).

TT Routines

The section gives detailed description of TT routines, their syntax, parameters and values they return. Double-precision and single-precision versions of the same routine are described together.

TT routines call Intel® MKL FFT interface (described in section "FFT Functions" in chapter "Fast Fourier Transforms"), which enhances performance of the routines.

`?_init_trig_transform`

Initializes basic data structures of a Trigonometric Transform.

Syntax

```c
void d_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_init_trig_transform(MKL_INT *n, MKL_INT *tt_type, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

• `mkl.h`

Input Parameters

- `n` MKL_INT*. Contains the size of the problem, which should be a positive integer greater than 1. Note that data vector of the transform, which other TT routines will use, must have size \( n+1 \) for all but staggered2 transforms. Staggered2 transforms require the vector of size \( n \).

- `tt_type` MKL_INT*. Contains the type of transform to compute, defined via a set of named constants. The following constants are available in the current implementation of TT interface: MKL_SINE_TRANSFORM, MKL_STAGGERED_SINE_TRANSFORM, MKL_STAGGERED2_SINE_TRANSFORM; MKL_COSINE_TRANSFORM, MKL_STAGGERED_COSINE_TRANSFORM, MKL_STAGGERED2_COSINE_TRANSFORM.
**Output Parameters**

**ipar**
MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.

**dpar**
double array of size $5n/2+2$. Contains double-precision data needed for Trigonometric Transform computations.

**spar**
float array of size $5n/2+2$. Contains single-precision data needed for Trigonometric Transform computations.

**stat**
MKL_INT*. Contains the routine completion status, which is also written to ipar[6]. The status should be 0 to proceed to other TT routines.

**Description**

The ?_init_trig_transform routine initializes basic data structures for Trigonometric Transforms of appropriate precision. After a call to ?_init_trig_transform, all subsequently invoked TT routines use values of ipar and dpar(spar) array parameters returned by ?_init_trig_transform. The routine initializes the entire array ipar. In the dpar or spar array, ?_init_trig_transform initializes elements that do not depend upon the type of transform. For a detailed description of arrays ipar, dpar and spar, refer to the Common Parameters section. You can skip a call to the initialization routine in your code. For more information, see Caveat on Parameter Modifications.

**Return Values**

- **stat= 0**
  The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this stat value.

- **stat= -99999**
  The routine failed to complete the task.

**d_commit_trig_transform**
Checks consistency and correctness of user’s data as well as initializes certain data structures required to perform the Trigonometric Transform.

**Syntax**

```c
void d_commit_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
```

```c
void s_commit_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

**Include Files**
- mkl.h

**Input Parameters**

- **f**
  double for d_commit_trig_transform,
  float for s_commit_trig_transform,
array of size \( n \) for staggered2 transforms and of size \( n+1 \) for all other transforms, where \( n \) is the size of the problem. Contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:

- \( f[0] \) and \( f[n] \) for sine transforms
- \( f[n] \) for staggered cosine transforms
- \( f[0] \) for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. These restrictions meet the requirements of the Intel® MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).

**ipar**

MKL_INT array of size 128. Contains integer data needed for Trigonometric Transform computations.

**dpar**

double array of size \( 5n/2+2 \). Contains double-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

**spar**

float array of size \( 5n/2+2 \). Contains single-precision data needed for Trigonometric Transform computations. The routine initializes most elements of this array.

**Output Parameters**

**handle**

DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms").

**ipar**

Contains integer data needed for Trigonometric Transform computations. On output, \( \text{ipar}[6] \) is updated with the \( \text{stat} \) value.

**dpar**

Contains double-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

**spar**

Contains single-precision data needed for Trigonometric Transform computations. On output, the entire array is initialized.

**stat**

MKL_INT*. Contains the routine completion status, which is also written to \( \text{ipar}[6] \).

**Description**

The routine \(?_commit_trig_transform\) checks consistency and correctness of the parameters to be passed to the transform routines \(?_forward_trig_transform\) and/or \(?_backward_trig_transform\). The routine also initializes the following data structures: \( \text{handle} \), \( \text{dpar} \) in case of \( d\_commit_trig_transform \), and \( \text{spar} \) in case of \( s\_commit_trig_transform \). The \(?_commit_trig_transform\) routine initializes only those elements of \( \text{dpar} \) or \( \text{spar} \) that depend upon the type of transform, defined in the \(?_init_trig_transform\) routine and passed to \(?_commit_trig_transform\) with the \( \text{ipar} \) array. The size of the problem \( n \), which determines sizes of the array parameters, is also passed to the routine with the \( \text{ipar} \) array and defined in the previously called \(?_init_trig_transform\) routine. For a detailed description of arrays \( \text{ipar}, \text{dpar} \) and \( \text{spar} \), refer to the Common Parameters section. The routine performs only a basic check for correctness and
consistency of the parameters. If you are going to modify parameters of TT routines, see the Caveat on Parameter Modifications section. Unlike `_init_trig_transform`, you must call the `_commit_trig_transform` routine in your code.

**Return Values**

stat = 11

The routine produced some warnings and made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

stat = 10

The routine made some changes in the parameters to achieve their correctness and/or consistency. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

stat = 1

The routine produced some warnings. You may proceed with computations by assigning `ipar[6]=0` if you are sure that the parameters are correct.

stat = 0

The routine completed the task normally.

stat = -100

The routine stopped for any of the following reasons:

- An error in the user’s data was encountered.
- Data in `ipar`, `dpar` or `spar` parameters became incorrect and/or inconsistent as a result of modifications.

stat = -1000

The routine stopped because of an FFT interface error.

stat = -10000

The routine stopped because the initialization failed to complete or the parameter `ipar[0]` was altered by mistake.

**NOTE**

Although positive values of `stat` usually indicate minor problems with the input data and Trigonometric Transform computations can be continued, you are highly recommended to investigate the problem first and achieve `stat=0`.

__`forward_trig_transform`__

*Computes the forward Trigonometric Transform of type specified by the parameter.*

**Syntax**

```c
void d_forward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_forward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

**Include Files**

- `mkl.h`
Input Parameters

\( f \)  
\[ \text{double for } \text{d_forward_trig_transform}, \]
\[ \text{float for } \text{s_forward_trig_transform}, \]
array of size \( n \) for staggered2 transforms and of size \( n+1 \) for all other transforms, where \( n \) is the size of the problem. On input, contains data vector to be transformed. Note that the following values should be 0.0 up to rounding errors:
- \( f[0] \) and \( f[n] \) for sine transforms
- \( f[n] \) for staggered cosine transforms
- \( f[0] \) for staggered sine transforms.
Otherwise, the routine will produce a warning, and the result of the computations for sine transforms may be wrong. The above restrictions meet the requirements of the Intel® MKL Poisson Solver, which the TT interface is primarily designed for (for details, see Fast Poisson Solver Routines).

\( \text{handle} \)  
\[ \text{DFTI_DESCRIPTOR_HANDLE*}. \text{The data structure used by Intel® MKL FFT interface (for details, see FFT Functions)}. \]

\( \text{ipar} \)  
\[ \text{MKL_INT} \text{ array of size } 128. \text{Contains integer data needed for Trigonometric Transform computations}. \]

\( \text{dpar} \)  
\[ \text{double array of size } 5n/2+2. \text{Contains double-precision data needed for Trigonometric Transform computations}. \]

\( \text{spar} \)  
\[ \text{float array of size } 5n/2+2. \text{Contains single-precision data needed for Trigonometric Transform computations}. \]

Output Parameters

\( f \)  
Contains the transformed vector on output.

\( \text{ipar} \)  
Contains integer data needed for Trigonometric Transform computations. On output, \( \text{ipar}[6] \) is updated with the \( \text{stat} \) value.

\( \text{stat} \)  
\[ \text{MKL_INT*}. \text{Contains the routine completion status, which is also written to } \text{ipar}[6]. \]

Description

The routine computes the forward Trigonometric Transform of type defined in the \_init_trig_transform routine and passed to \_forward_trig_transform with the \text{ipar} array. The size of the problem \( n \), which determines sizes of the array parameters, is also passed to the routine with the \text{ipar} array and defined in the previously called \_init_trig_transform routine. The other data that facilitates the computation is created by \_commit_trig_transform and supplied in \text{dpar} or \text{spar}. For a detailed description of arrays \text{ipar}, \text{dpar} and \text{spar}, refer to the Common Parameters section. The routine has a commit step, which calls the \_commit_trig_transform routine. The transform is computed according to formulas given in the Transforms Implemented section. The routine replaces the input vector \( f \) with the transformed vector.
NOTE
If you need a copy of the data vector $f$ to be transformed, make the copy before calling
the $?_forward_trig_transform$ routine.

Return Values

$stat = 0$
The routine completed the task normally.

$stat = -100$
The routine stopped for any of the following reasons:
• An error in the user’s data was encountered.
• Data in $ipar$, $dpar$ or $spar$ parameters became
  incorrect and/or inconsistent as a result of
  modifications.

$stat = -1000$
The routine stopped because of an FFT interface error.

$stat = -10000$
The routine stopped because its commit step failed to
complete or the parameter $ipar[0]$ was altered by
mistake.

$?_backward_trig_transform$
Computes the backward Trigonometric Transform of
type specified by the parameter.

Syntax

```c
void d_backward_trig_transform(double f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], double dpar[], MKL_INT *stat);
void s_backward_trig_transform(float f[], DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], float spar[], MKL_INT *stat);
```

Include Files

• mkl.h

Input Parameters

$f$

double for $d_backward_trig_transform$,
float for $s_backward_trig_transform$,
array of size $n$ for staggered2 transforms and of size $n+1$ for all other
transforms, where $n$ is the size of the problem. On input, contains
data vector to be transformed. Note that the following values should
be 0.0 up to rounding errors:
• $f[0]$ and $f[n]$ for sine transforms
• $f[n]$ for staggered cosine transforms
• $f[0]$ for staggered sine transforms.

Otherwise, the routine will produce a warning, and the result of the
computations for sine transforms may be wrong. The above
restrictions meet the requirements of the Intel® MKL Poisson Solver,
which the TT interface is primarily designed for (for details, see Fast
Poisson Solver Routines).
The routine computes the backward Trigonometric Transform of type defined in the \texttt{?_init_trig_transform} routine and passed to \texttt{?_backward_trig_transform} with the \texttt{ipar} array. The size of the problem \( n \), which determines sizes of the array parameters, is also passed to the routine with the \texttt{ipar} array and defined in the previously called \texttt{?_init_trig_transform} routine. The other data that facilitates the computation is created by \texttt{?_commit_trig_transform} and supplied in \texttt{dpar} or \texttt{spar}. For a detailed description of arrays \texttt{ipar}, \texttt{dpar} and \texttt{spar}, refer to the \texttt{Common Parameters} section. The routine has a commit step, which calls the \texttt{?_commit_trig_transform} routine. The transform is computed according to formulas given in the \texttt{Transforms Implemented} section. The routine replaces the input vector \( f \) with the transformed vector.

\textbf{NOTE}

If you need a copy of the data vector \( f \) to be transformed, make the copy before calling the \texttt{?_backward_trig_transform} routine.

\textbf{Return Values}

- \texttt{stat} = 0: The routine completed the task normally.
- \texttt{stat} = -100: The routine stopped for any of the following reasons:
  - An error in the user's data was encountered.
  - Data in \texttt{ipar}, \texttt{dpar} or \texttt{spar} parameters became incorrect and/or inconsistent as a result of modifications.
- \texttt{stat} = -1000: The routine stopped because of an FFT interface error.
- \texttt{stat} = -10000: The routine stopped because its commit step failed to complete or the parameter \texttt{ipar[0]} was altered by mistake.
free_trig_transform
Cleans the memory allocated for the data structure
used by the FFT interface.

Syntax
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *handle, MKL_INT ipar[], MKL_INT *stat);

Include Files
• mkl.h

Input Parameters

ipar MKL_INT array of size 128. Contains integer data needed for
Trigonometric Transform computations.

handle DFTI_DESCRIPTOR_HANDLE*. The data structure used by Intel® MKL
FFT interface (for details, refer to section "FFT Functions" in chapter
"Fast Fourier Transforms").

Output Parameters

handle The data structure used by Intel® MKL FFT interface. Memory allocated
for the structure is released on output.

ipar Contains integer data needed for Trigonometric Transform
computations. On output, ipar[6] is updated with the stat value.

stat MKL_INT*. Contains the routine completion status, which is also
written to ipar[6].

Description
The free_trig_transform routine cleans the memory used by the handle structure, needed for Intel® MKL
FFT functions. To release the memory allocated for other parameters, include cleaning of the memory in your
code.

Return Values

stat= 0 The routine completed the task normally.
stat= -1000 The routine stopped because of an FFT interface error.
stat= -99999 The routine failed to complete the task.

Common Parameters of the Trigonometric Transforms
This section provides description of array parameters that hold TT routine options: ipar, dpar and spar.

NOTE
Initial values are assigned to the array parameters by the appropriate _init_trig_transform
and _commit_trig_transform routines.
**Elements of the ipar Array**

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Contains the size of the problem to solve. The <code>?_init_trig_transform</code> routine sets <code>ipar[0]=n</code>, and all subsequently called TT routines use <code>ipar[0]</code> as the size of the transform.</td>
</tr>
</tbody>
</table>
| 1     | Contains error messaging options:  
  - `ipar[1]=-1` indicates that all error messages will be printed to the file `MKL_Trig_Transforms_log.txt` in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device.  
  - `ipar[1]=0` indicates that no error messages will be printed.  
  - `ipar[1]=1` (default) indicates that all error messages will be printed to the preconnected default output device (usually, screen).  
  In case of errors, each TT routine assigns a non-zero value to `stat` regardless of the `ipar[1]` setting. |
| 2     | Contains warning messaging options:  
  - `ipar[2]=-1` indicates that all warning messages will be printed to the file `MKL_Trig_Transforms_log.txt` in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device.  
  - `ipar[2]=0` indicates that no warning messages will be printed.  
  - `ipar[2]=1` (default) indicates that all warning messages will be printed to the preconnected default output device (usually, screen).  
  In case of warnings, the `stat` parameter will acquire a non-zero value regardless of the `ipar[2]` setting. |
| 3 through 4 | Reserved for future use. |
| 5     | Contains the type of the transform. The `?_init_trig_transform` routine sets `ipar[5]=tt_type`, and all subsequently called TT routines use `ipar[5]` as the type of the transform. |
| 6     | Contains the `stat` value returned by the last completed TT routine. Used to check that the previous call to a TT routine completed with `stat=0`. |
| 7     | Informs the `?_commit_trig_transform` routines whether to initialize data structures `dpar`, `spar`, and `handle`. `ipar[7]=0` indicates that the routine should skip the initialization and only check correctness and consistency of the parameters. Otherwise, the routine initializes the data structures. The default value is 1.  
  The possibility to check correctness and consistency of input data without initializing data structures `dpar`, `spar` and `handle` enables avoiding performance losses in a repeated use of the same transform for different data vectors. Note that you can benefit from the opportunity that `ipar[7]` gives only if you are sure to have supplied proper tolerance value in the `dpar` or `spar` array. Otherwise, avoid tuning this parameter. |
| 8     | Contains message style options for TT routines. Specifically, if `ipar[8]` is non-zero, TT routines print the messages in C-style notations. The default value is 1. |
Index | Description
---|---
0 | Contains the first absolute tolerance used by the appropriate _commit_trig_transform routine. For a staggered cosine or a sine transform, \( f[n] \) should be equal to 0.0 and for a staggered sine or a sine transform, \( f[0] \) should be equal to 0.0. The _commit_trig_transform routine checks whether absolute values of these parameters are below \( dpar[0]*n \) or \( spar[0]*n \), depending on the routine precision. To suppress warnings resulting from tolerance checks, set \( dpar[0] \) or \( spar[0] \) to a sufficiently large number.

1 | Reserved for future use.

2 through \( 5n/2+1 \) | Contain tabulated values of trigonometric functions. Contents of the elements depend upon the type of transform \( tt\_type \), set up in the _commit_trig_transform routine:
- If \( tt\_type=MKL\_SINE\_TRANSFORM \), the transform uses only the first \( n/2 \) array elements, which contain tabulated sine values.
- If \( tt\_type=MKL\_STAGGERED\_SINE\_TRANSFORM \), the transform uses only the first \( 3n/2 \) array elements, which contain tabulated sine and cosine values.
- If \( tt\_type=MKL\_STAGGERED2\_SINE\_TRANSFORM \), the transform uses all the \( 5n/2 \) array elements, which contain tabulated sine and cosine values.
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• If <code>tt_type=MKL_COSINE_TRANSFORM</code>, the transform uses only the first $n$ array elements, which contain tabulated cosine values.</td>
<td></td>
</tr>
<tr>
<td>• If <code>tt_type=MKL_STAGGERED_COSINE_TRANSFORM</code>, the transform uses only the first $3n/2$ elements, which contain tabulated sine and cosine values.</td>
<td></td>
</tr>
<tr>
<td>• If <code>tt_type=MKL_STAGGERED2_COSINE_TRANSFORM</code>, the transform uses all the $5n/2$ elements, which contain tabulated sine and cosine values.</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE**
To save memory, you can define the array size depending upon the type of transform.

**Caveat on Parameter Modifications**

Flexibility of the TT interface enables you to skip a call to the `_init_trig_transform` routine and to initialize the basic data structures explicitly in your code. You may also need to modify the contents of `ipar`, `dpar` and `spar` arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or wrong computation. You can perform a basic check for correctness and consistency of parameters by calling the `_commit_trig_transform` routine; however, this does not ensure the correct result of a transform but only reduces the chance of errors or wrong results.

**NOTE**
To supply correct and consistent parameters to TT routines, you should have considerable experience in using the TT interface and good understanding of elements that the `ipar`, `spar` and `dpar` arrays contain and dependencies between values of these elements.

However, in rare occurrences, even advanced users might fail to compute a transform using TT routines after the parameter modifications. In cases like these, refer for technical support at [http://www.intel.com/software/products/support/](http://www.intel.com/software/products/support/).

**WARNING**
The only way that ensures proper computation of the Trigonometric Transforms is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of `ipar`, `dpar` and `spar` arrays unless a strong need arises.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Trigonometric Transform Implementation Details

Several aspects of the Intel® MKL TT interface are platform-specific and language-specific. To promote portability across platforms and ease of use across different languages, Intel® MKL provides you with the TT language-specific header file to include in your code:

- mkl_trig_transforms.h, to be used together with mkl_dfti.h.

**NOTE**

- Use of the Intel® MKL TT software without including the above language-specific header files is not supported.

Header File

The header file below defines the following function prototypes:

```c
void d_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, double *, MKL_INT *);
void d_commit_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_forward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void d_backward_trig_transform(double *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, double *, MKL_INT *);
void s_init_trig_transform(MKL_INT *, MKL_INT *, MKL_INT *, float *, MKL_INT *);
void s_commit_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_forward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void s_backward_trig_transform(float *, DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, float *, MKL_INT *);
void free_trig_transform(DFTI_DESCRIPTOR_HANDLE *, MKL_INT *, MKL_INT *);
```

Fast Poisson Solver Routines

In addition to the Real Discrete Trigonometric Transforms (TT) interface (refer to Trigonometric Transform Routines), Intel® MKL supports the Poisson Solver interface. This interface implements a group of routines (Poisson Solver routines) used to compute a solution of Laplace, Poisson, and Helmholtz problems of a special kind using discrete Fourier transforms. Laplace and Poisson problems are special cases of a more general Helmholtz problem. The problems that are solved by the Poisson Solver interface are defined more exactly in the Poisson Solver Implementation subsection. The Poisson Solver interface provides much flexibility of use: you can call routines with the default parameter values or adjust routines to your particular needs by manually tuning routine parameters. You can adjust the style of error and warning messages to a C notation by setting up a dedicated parameter. This adds convenience to debugging, because you can read information in the way that is natural for your code. The Intel® MKL Poisson Solver interface currently contains only routines that implement the following solvers:

- Fast Laplace, Poisson and Helmholtz solvers in a Cartesian coordinate system
- Fast Poisson and Helmholtz solvers in a spherical coordinate system.

Poisson Solver Implementation

Poisson Solver routines enable approximate solving of certain two-dimensional and three-dimensional problems. Figure "Structure of the Poisson Solver" shows the general structure of the Poisson Solver.
Structure of the Poisson Solver

Sections below provide details of the problems that can be solved using Intel® MKL Poisson Solver.

Two-Dimensional Problems

Notational Conventions

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_x < x < b_x$, $a_y < y < b_y$ on a Cartesian plane:

- $bd_{a_x} = \{x = a_x, a_y \leq y \leq b_y\}$, $bd_{b_x} = \{x = b_x, a_y \leq y \leq b_y\}$
- $bd_{a_y} = \{a_x \leq x \leq b_x, y = a_y\}$, $bd_{b_y} = \{a_x \leq x \leq b_x, y = b_y\}$.

The following figure shows these boundaries:
The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y$, so $bd_+$ denotes any of the above boundaries.

The Poisson Solver interface description uses the following notation for boundaries of a rectangular domain $a_\phi < \phi < b_\phi, a_\theta < \theta < b_\theta$ on a sphere $0 \leq \phi \leq 2\pi, 0 \leq \theta \leq \pi$:

- $bd_a\phi = \{\phi = a_\phi, a_\theta \leq \theta \leq b_\theta\}$
- $bd_b\phi = \{\phi = b_\phi, a_\theta \leq \theta \leq b_\theta\}$
- $bd_a\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = a_\theta\}$
- $bd_b\theta = \{a_\phi \leq \phi \leq b_\phi, \theta = b_\theta\}$

The wildcard "~" may stand for any of the symbols $a_\phi, b_\phi, a_\theta, b_\theta$, so $bd_\sim$ denotes any of the above boundaries.

**Two-dimensional Helmholtz problem on a Cartesian plane**

The two-dimensional (2D) Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} + qu = f(x,y), \quad q = \text{const} \geq 0$$

in a rectangle, that is, a rectangular domain $a_x < x < b_x, a_y < y < b_y$, with one of the following boundary conditions on each boundary $bd_+$:

- The Dirichlet boundary condition
  $$u(x,y) = G(x,y)$$
- The Neumann boundary condition
  $$\frac{\partial u}{\partial n}(x,y) = g(x,y)$$

where
- $n = -x$ on $bd_a_x$
- $n = x$ on $bd_b_x$
- $n = -y$ on $bd_a_y$
- $n = y$ on $bd_b_y$.

- Periodic boundary conditions
Two-dimensional Poisson problem on a Cartesian plane

The Poisson problem is a special case of the Helmholtz problem, when \( q = 0 \). The 2D Poisson problem is to find an approximate solution of the Poisson equation

\[
-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} = f(x, y)
\]

in a rectangle \( a_x < x < b_x, a_y < y < b_y \) with the Dirichlet, Neumann, or periodic boundary conditions on each boundary \( bd_+ \). In case of a problem with the Neumann boundary condition on the entire boundary, you can find the solution of the problem only up to a constant. In this case, the Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

Two-dimensional (2D) Laplace problem on a Cartesian plane

The Laplace problem is a special case of the Helmholtz problem, when \( q = 0 \) and \( f(x, y) = 0 \). The 2D Laplace problem is to find an approximate solution of the Laplace equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0
\]

in a rectangle \( a_x < x < b_x, a_y < y < b_y \) with the Dirichlet, Neumann, or periodic boundary conditions on each boundary \( bd_+ \).

Helmholtz problem on a sphere

The Helmholtz problem on a sphere is to find an approximate solution of the Helmholtz equation

\[-\Delta_s u + qu = f, \quad q = \text{const} \geq 0,\]

\[
\Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial^2}{\partial \phi^2} + \frac{1}{\sin \theta} \frac{\partial}{\partial \theta} \left( \sin \theta \frac{\partial}{\partial \theta} \right)
\]

in a domain bounded by angles \( a_\phi \leq \phi \leq b_\phi, a_\theta \leq \theta \leq b_\theta \) (spherical rectangle), with boundary conditions for particular domains listed in Table “Details of Helmholtz Problem on a Sphere”.

Details of Helmholtz Problem on a Sphere

<table>
<thead>
<tr>
<th>Domain on a sphere</th>
<th>Boundary condition</th>
<th>Periodic/non-periodic case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rectangular, that is, ( b_\phi - a_\phi &lt; 2\pi ) and ( b_\theta - a_\theta &lt; \pi )</td>
<td>Homogeneous Dirichlet boundary conditions on each boundary ( bd_+ \sim )</td>
<td>non-periodic</td>
</tr>
<tr>
<td>Where ( a_\phi = 0, b_\phi = 2\pi ), and ( b_\theta - a_\theta &lt; \pi )</td>
<td>Homogeneous Dirichlet boundary conditions on the boundaries ( bd_a \theta ) and ( bd_b \theta )</td>
<td>periodic</td>
</tr>
<tr>
<td>Entire sphere, that is, ( a_\phi = 0, b_\phi = 2\pi ), ( a_\theta = 0 ), and ( b_\theta = \pi )</td>
<td>Boundary condition ( \sin \theta \frac{\partial u}{\partial \theta} = 0 ) ( \theta \to 0 ) ( \theta \to \pi ) at the poles</td>
<td>periodic</td>
</tr>
</tbody>
</table>

Poisson problem on a sphere
The Poisson problem is a special case of the Helmholtz problem, when \( q = 0 \). The Poisson problem on a sphere is to find an approximate solution of the Poisson equation

\[
-\Delta s u = f, \quad \Delta_s = \frac{1}{\sin^2 \theta} \frac{\partial}{\partial \phi} \left( \sin \theta \frac{\partial}{\partial \theta} \right)
\]

in a spherical rectangle \( a_{\phi} \leq \phi \leq b_{\phi}, a_{\theta} \leq \theta \leq b_{\theta} \) in cases listed in Table "Details of Helmholtz Problem on a Sphere". The solution to the Poisson problem on the entire sphere can be found up to a constant only. In this case, Poisson Solver will compute the solution that provides the minimal Euclidean norm of a residual.

**Approximation of 2D problems**

To find an approximate solution for any of the 2D problems, in the rectangular domain a uniform mesh can be defined for the Cartesian case as:

\[
\begin{align*}
x_i &= a_x + i h_x, \quad y_j = a_y + j h_y, \\
i &= 0,..., n_x, \quad j = 0,..., n_y, \quad h_x = \frac{b_x - a_x}{n_x}, \quad h_y = \frac{b_y - a_y}{n_y}
\end{align*}
\]

and for the spherical case as:

\[
\begin{align*}
\phi_i &= a_{\phi} + i h_{\phi}, \quad \theta_j = a_{\theta} + j h_{\theta}, \\
i &= 0,..., n_{\phi}, \quad j = 0,..., n_{\theta}, \quad h_{\phi} = \frac{b_{\phi} - a_{\phi}}{n_{\phi}}, \quad h_{\theta} = \frac{b_{\theta} - a_{\theta}}{n_{\theta}}.
\end{align*}
\]

The Poisson Solver uses the standard five-point finite difference approximation on this mesh to compute the approximation to the solution:

- In the Cartesian case, the values of the approximate solution will be computed in the mesh points \((x_i, y_j)\) provided that you can supply the values of the right-hand side \( f(x, y) \) in these points and the values of the appropriate boundary functions \( G(x, y) \) and/or \( g(x,y) \) in the mesh points laying on the boundary of the rectangular domain.
- In the spherical case, the values of the approximate solution will be computed in the mesh points \((\phi_i, \theta_j)\) provided that you can supply the values of the right-hand side \( f(\phi, \theta) \) in these points.

**NOTE**

The number of mesh intervals \( n_{\phi} \) in the \( \phi \) direction of a spherical mesh must be even in the periodic case. The Poisson Solver does not support spherical meshes that do not meet this condition.

**Three-Dimensional Problems**

**Notational Conventions**

The Poisson Solver interface description uses the following notation for boundaries of a parallelepiped domain \( a_x < x < b_x, a_y < y < b_y, a_z < z < b_z \):

- \( \text{bd}_a x = \{ x = a_x, \quad a_y \leq y \leq b_y, \quad a_z \leq z \leq b_z \} \),
- \( \text{bd}_b x = \{ x = b_x, \quad a_y \leq y \leq b_y, \quad a_z \leq z \leq b_z \} \),
- \( \text{bd}_a y = \{ a_x \leq x \leq b_x, \quad y = a_y, \quad a_z \leq z \leq b_z \} \),
- \( \text{bd}_b y = \{ a_x \leq x \leq b_x, \quad y = b_y, \quad a_z \leq z \leq b_z \} \),
- \( \text{bd}_a z = \{ a_x \leq x \leq b_x, \quad a_y \leq y \leq b_y, \quad z = a_z \} \),
- \( \text{bd}_b z = \{ a_x \leq x \leq b_x, \quad a_y \leq y \leq b_y, \quad z = b_z \} \).

The following figure shows these boundaries:
The wildcard "+" may stand for any of the symbols $a_x, b_x, a_y, b_y, a_z, b_z$, so $bd_+$ denotes any of the above boundaries.

**Three-dimensional (3D) Helmholtz problem**

The 3D Helmholtz problem is to find an approximate solution of the Helmholtz equation

$$-\frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} + qu = f(x, y, z), \ q = \text{const } \geq 0$$

in a parallelepiped, that is, a parallelepiped domain $a_x < x < b_x, a_y < y < b_y, a_z < z < b_z$, with one of the following boundary conditions on each boundary $bd_+$:

- **The Dirichlet boundary condition**
  
  $$u(x, y, z) = G(x, y, z)$$

- **The Neumann boundary condition**

  $$\frac{\partial u}{\partial n}(x, y, z) = g(x, y, z)$$

  where

  $n = -x$ on $bd_{a_x}$, $n = x$ on $bd_{b_x}$,

  $n = -y$ on $bd_{a_y}$, $n = y$ on $bd_{b_y}$,

  $n = -z$ on $bd_{a_z}$, $n = z$ on $bd_{b_z}$.

- **Periodic boundary conditions**

  $$u(a_x, y, z) = u(b_x, y, z), \ \frac{\partial}{\partial x} u(a_x, y, z) = \frac{\partial}{\partial x} u(b_x, y, z),$$

  $$u(a_y, x, z) = u(b_y, x, z), \ \frac{\partial}{\partial y} u(a_y, x, z) = \frac{\partial}{\partial y} u(b_y, x, z),$$

  $$u(a_z, x, y) = u(b_z, x, y), \ \frac{\partial}{\partial z} u(a_z, x, y) = \frac{\partial}{\partial z} u(b_z, x, y).$$
Three-dimensional (3D) Poisson problem

The Poisson problem is a special case of the Helmholtz problem, when \( q = 0 \). The 3D Poisson problem is to find an approximate solution of the Poisson equation

\[
- \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z)
\]

in a parallelepiped \( a_x < x < b_x, a_y < y < b_y, a_z < z < b_z \) with the Dirichlet, Neumann, or periodic boundary conditions on each boundary \( bd_+ \).

Three-dimensional (3D) Laplace problem

The Laplace problem is a special case of the Helmholtz problem, when \( q = 0 \) and \( f(x, y, z) = 0 \). The 3D Laplace problem is to find an approximate solution of the Laplace equation

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0
\]

in a parallelepiped \( a_x < x < b_x, a_y < y < b_y, a_z < z < b_z \) with the Dirichlet, Neumann, or periodic boundary conditions on each boundary \( bd_+ \).

Approximation of 3D problems

To find an approximate solution for each of the 3D problems, a uniform mesh can be defined in the parallelepipied domain as:

\[
\{ x_i = a_x + i h_x, y_j = a_y + j h_y, z_k = a_z + j h_z \}
\]

where

\[
i = 0, ..., n_x, j = 0, ..., n_y, k = 0, ..., n_z,
\]

\[
h_x = \frac{b_x - a_x}{n_x}, h_y = \frac{b_y - a_y}{n_y}, h_z = \frac{b_z - a_z}{n_z}.
\]

The Poisson Solver uses the standard seven-point finite difference approximation on this mesh to compute the approximation to the solution. The values of the approximate solution will be computed in the mesh points \( (x_i, y_j, z_k) \), provided that you can supply the values of the right-hand side \( f(x, y, z) \) in these points and the values of the appropriate boundary functions \( G(x, y, z) \) and/or \( g(x, y, z) \) in the mesh points laying on the boundary of the parallelepiped domain.

Sequence of Invoking Poisson Solver Routines

**NOTE**

This description always shows the solution process for the Helmholtz problem, because Fast Poisson Solvers and Fast Laplace Solvers are special cases of Fast Helmholtz Solvers (see Poisson Solver Implementation).
The Poisson Solver interface enables you to compute a solution of the Helmholtz problem in four steps. Each step is performed by a dedicated routine. Table "Poisson Solver Interface Routines" lists the routines and briefly describes their purpose.

Most Poisson Solver routines have versions operating with single-precision and double-precision data. Names of such routines begin respectively with "s" and "d". The wildcard "?" stands for either of these symbols in routine names. The routines for the Cartesian coordinate system have 2D and 3D versions. Their names end respectively in "2D" and "3D". The routines for spherical coordinate system have periodic and non-periodic versions. Their names end respectively in "p" and "np".

### Poisson Solver Interface Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np</td>
<td>Initializes basic data structures for Fast Helmholtz Solver in the 2D/3D/periodic/non-periodic case, respectively.</td>
</tr>
<tr>
<td>?_commit_Helmholtz_2D/?_commit_Helmholtz_3D/?_commit_sph_p/?_commit_sph_np</td>
<td>Checks consistency and correctness of input data and initializes data structures for the solver, including those used by the Intel® MKL FFT interface.</td>
</tr>
<tr>
<td>?_Helmholtz_2D/?_Helmholtz_3D/?_sph_p/?_sph_np</td>
<td>Computes an approximate solution of the 2D/3D/periodic/non-periodic Helmholtz problem (see Poisson Solver Implementation) specified by the parameters.</td>
</tr>
<tr>
<td>free_Helmholtz_2D/free_Helmholtz_3D/free_sph_p/free_sph_np</td>
<td>Releases the memory used by the data structures needed for calling the Intel® MKL FFT interface.</td>
</tr>
</tbody>
</table>

Poisson Solver routines call the Intel® MKL FFT interface for better performance.

To find an approximate solution of Helmholtz problem only once, the Intel® MKL Poisson Solver interface routines are normally invoked in the order in which they are listed in Table "Poisson Solver Interface Routines".

**NOTE**

Though the order of invoking Poisson Solver routines may be changed, it is highly recommended to follow the above order of routine calls.

The diagram in Figure "Typical Order of Invoking Poisson Solver Routines" indicates the typical order in which Poisson Solver routines can be invoked in a general case.
Typical Order of Invoking Poisson Solver Routines

1. Initialize
2. Change routine parameters (manually)
3. Commit
4. Fast Helmholtz Solver
5. Free memory

A general scheme of using Poisson Solver routines for double-precision computations in a 3D Cartesian case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below from the 3D to 2D case by changing the ending of the Poisson Solver routine names.

```c
...  
   d_init_Helmholtz_3D(ax, bx, ay, by, az, bz, nx, ny, nz, BCtype, ipar, dpar, &stat);  
   /* change parameters in ipar and/or dpar if necessary. */  
   /* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data 
   that is stored in f, save it to another location before the function call below */  
   d_commit_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, 
                       dpar, &stat);  
   d_Helmholtz_3D(f, bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz, &xhandle, &yhandle, ipar, 
                       dpar, &stat);  
   free_Helmholtz_3D(&xhandle, &yhandle, ipar, &stat);  
   /* here you may clean the memory used by f, dpar, ipar */  
...  
```

A general scheme of using Poisson Solver routines for double-precision computations in a spherical periodic case is shown below. You can change this scheme to a scheme for single-precision computations by changing the initial letter of the Poisson Solver routine names from "d" to "s". You can also change the scheme below to a scheme for a non-periodic case by changing the ending of the Poisson Solver routine names from "p" to "np".

```c
...  
   d_init_sph_p(ap, bp, at, bt, np, nt, &q, ipar, dpar, &stat);  
   /* change parameters in ipar and/or dpar if necessary. */  
   /* note that the result of the Fast Helmholtz Solver will be in f. If you want to keep the data 
   that is stored in f, save it to another location before the function call below */  
   d_commit_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);  
   d_sph_p(f, &handle_s, &handle_c, ipar, dpar, &stat);  
   free_sph_p(&handle_s, &handle_c, ipar, &stat);  
   /* here you may clean the memory used by f, dpar, ipar */  
...  
```

You can find examples of code that uses Poisson Solver routines to solve Helmholtz problem (in both Cartesian and spherical cases) in the examples\pdepoissonc\source folder in your Intel® MKL directory.
Fast Poisson Solver Interface Description

All numerical types in this section are either standard C types float and double or MKL_INT integer type.
For more information on the C types, refer to C Datatypes Specific to Intel® MKL and the Intel® MKL Developer Guide. To better understand usage of the types, see examples in the examples\pdepoissonc\source folder in your Intel® MKL directory.

Routine Options

All Poisson Solver routines use parameters for passing various options to the routines. These parameters are arrays ipar, dpar, and spar. Values for these parameters should be specified very carefully (see Common Parameters). You can change these values during computations to meet your needs. For more details, see the descriptions of specific routines.

WARNING
To avoid failure or incorrect results, you must provide correct and consistent parameters to the routines.

User Data Arrays

Poisson Solver routines take arrays of user data as input. For example, the d_Helmholtz_3D routine takes user arrays to compute an approximate solution to the 3D Helmholtz problem. To minimize storage requirements and improve the overall run-time efficiency, Intel® MKL Poisson Solver routines do not make copies of user input arrays.

NOTE
If you need a copy of your input data arrays, you must save them yourself.

For better performance, align your data arrays as recommended in the Intel® MKL Developer Guide (search the document for coding techniques to improve performance).

Routines for the Cartesian Solver

The section describes Poisson Solver routines for the Cartesian case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and 2D and 3D.

NOTE
Some of the routine parameters are used only in the 3D Fast Helmholtz Solver.

Poisson Solver routines call Intel® MKL FFT routines (described in section "FFT Functions" in chapter "Fast Fourier Transforms"), which enhance performance of the Poisson Solver routines.

?_init_Helmholtz_2D/?_init_Helmholtz_3D
Initializes basic data structures of the Fast 2D/3D Helmholtz Solver.

Syntax

void d_init_Helmholtz_2D (const double * ax, const double * bx, const double * ay, const double * by, const MKL_INT * nx, const MKL_INT * ny, const char * BCtype, const double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat);
void s_init_Helmholtz_2D(const float * ax, const float * bx, const float * ay, const float * by, const MKL_INT * nx, const MKL_INT * ny, const char * BCTYPE, const float * q, MKL_INT * ipar, float * spar, MKL_INT * stat);

void d_init_Helmholtz_3D(const double * ax, const double * bx, const double * ay, const double * by, const double * az, const double * bz, const MKL_INT * nx, const MKL_INT * ny, const MKL_INT * nz, const char * BCTYPE, const double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat);

void s_init_Helmholtz_3D(const float * ax, const float * bx, const float * ay, const float * by, const float * az, const float * bz, const MKL_INT * nx, const MKL_INT * ny, const MKL_INT * nz, const char * BCTYPE, const float * q, MKL_INT * ipar, float * spar, MKL_INT * stat);

Include Files

- mkl.h

Input Parameters

**ax**

double* for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.
The coordinate of the leftmost boundary of the domain along the x-axis.

**bx**

double* for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.
The coordinate of the rightmost boundary of the domain along the x-axis.

**ay**

double* for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.
The coordinate of the leftmost boundary of the domain along the y-axis.

**by**

double* for `d_init_Helmholtz_2D/d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_2D/s_init_Helmholtz_3D`.
The coordinate of the rightmost boundary of the domain along the y-axis.

**az**

double* for `d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_3D`.
The coordinate of the leftmost boundary of the domain along the z-axis. This parameter is needed only for the `s_init_Helmholtz_3D` routine.

**bz**

double* for `d_init_Helmholtz_3D`,
float* for `s_init_Helmholtz_3D`.
The coordinate of the rightmost boundary of the domain along the z-axis. This parameter is needed only for the `s_init_Helmholtz_3D` routine.
nx  MKL_INT*. The number of mesh intervals along the x-axis.

ny  MKL_INT*. The number of mesh intervals along the y-axis.

nz  MKL_INT*. The number of mesh intervals along the z-axis. This parameter is needed only for the \_init\_Helmholtz\_3D routine.

BCtype char*. Contains the type of boundary conditions on each boundary. Must contain four characters for \_init\_Helmholtz\_2D and six characters for \_init\_Helmholtz\_3D. Each of the characters can be 'N' (Neumann boundary condition), 'D' (Dirichlet boundary condition), or 'P' (periodic boundary conditions). Specify the types of boundary conditions for the boundaries in the following order: \(bd_a_x\), \(bd_b_x\), \(bd_a_y\), \(bd_b_y\), \(bd_a_z\), and \(bd_b_z\). Specify periodic boundary conditions on the respective boundaries in pairs (for example, 'PPDD' or 'NNPP' in the 2D case). The types of boundary conditions for the last two boundaries are needed only in the 3D case.

q  double* for d\_init\_Helmholtz\_2D/d\_init\_Helmholtz\_3D, float* for s\_init\_Helmholtz\_2D/s\_init\_Helmholtz\_3D. The constant Helmholtz coefficient. Note that to solve Poisson or Laplace problem, you should set the value of \(q\) to 0.

**Output Parameters**

*ipar*  MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to ipar).

*dpar*  double array of size 5\(\times nx/2+7\) in the 2D case or 5\(\times (nx+ny)/2+9\) in the 3D case. Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).

*spar*  float array of size 5\(\times nx/2+7\) in the 2D case or 5\(\times (nx+ny)/2+9\) in the 3D case. Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to dpar and spar).

*stat*  MKL_INT*. Routine completion status, which is also written to ipar[0]. Continue to call other Poisson Solver routines only if the status is 0.

**Description**

The \_init\_Helmholtz\_2D/\_init\_Helmholtz\_3D routines initialize basic data structures for Poisson Solver computations of the appropriate precision. All routines invoked after a call to a \_init\_Helmholtz\_2D/\_init\_Helmholtz\_3D routine use values of the ipar, dpar and spar array parameters returned by the routine. Detailed description of the array parameters can be found in Common Parameters.

**Caution**

Data structures initialized and created by 2D flavors of the routine cannot be used by 3D flavors of any Poisson Solver routines, and vice versa.
You can skip calls to these routines in your code. However, see Caveat on Parameter Modifications for information on initializing the data structures.

**Return Values**

\[ \text{stat} = 0 \]

The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this \( \text{stat} \) value.

\[ \text{stat} = -99999 \]

The routine failed to complete the task because of a fatal error.

**commit_Helmholtz_2D/?commit_Helmholtz_3D**

Checks consistency and correctness of input data and initializes certain data structures required to solve 2D/3D Helmholtz problem.

**Syntax**

void d_commit_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, double * dpar, MKL_INT * stat );

void s_commit_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, float * spar, MKL_INT * stat );

void d_commit_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, double * dpar, MKL_INT * stat );

void s_commit_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, float * spar, MKL_INT * stat );

**Include Files**

- mkl.h

**Input Parameters**

\[ f \]

- double* for d_commit_Helmholtz_2D/d_commit_Helmholtz_3D,
  - float* for s_commit_Helmholtz_2D/s_commit_Helmholtz_3D.

Contains the right-hand side of the problem packed in a single vector:

- 2D problem: The size of the vector for the is \((nx+1)*(ny+1)\). The value of the right-hand side in the mesh point \((i, j)\) is stored in \(f[i+j*(nx+1)]\).
- 3D problem: The size of the vector for the is \((nx+1)*(ny+1)*(nz+1)\). The value of the right-hand side in the mesh point \((i, j, k)\) is stored in \(f[i+j*(nx+1)+k*(nx+1)*(ny+1)]\).

Note that to solve the Laplace problem, you should set all the elements of the array \(f\) to 0.
Note also that the array $f$ may be altered by the routine. To preserve the $f$ vector, save it to another memory location.

$ipar$  
MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver (for details, refer to $ipar$).

$dparr$  
double array of size depending on the dimension of the problem:  
- 2D problem: $5*nx/2+7$  
- 3D problem: $5*(nx+ny)/2+9$

Contains double-precision data to be used by the Fast Helmholtz Solver (for details, refer to $dparr$ and $spar$).

$spar$  
float array of size depending on the dimension of the problem:  
- 2D problem: $5*nx/2+7$  
- 3D problem: $5*(nx+ny)/2+9$

Contains single-precision data to be used by the Fast Helmholtz Solver (for details, refer to $dparr$ and $spar$).

$bd_ax$  
double* for $d_commit_Helmholtz_2D/d_commit_Helmholtz_3D$,  
float* for $s_commit_Helmholtz_2D/s_commit_Helmholtz_3D$.

Contains values of the boundary condition on the leftmost boundary of the domain along the $x$-axis (for more information, refer to a detailed description of $bd_ax$).

$bd_bx$  
double* for $d_commit_Helmholtz_2D/d_commit_Helmholtz_3D$,  
float* for $s_commit_Helmholtz_2D/s_commit_Helmholtz_3D$.

Contains values of the boundary condition on the rightmost boundary of the domain along the $x$-axis (for more information, refer to a detailed description of $bd_bx$).

$bd_ay$  
double* for $d_commit_Helmholtz_2D/d_commit_Helmholtz_3D$,  
float* for $s_commit_Helmholtz_2D/s_commit_Helmholtz_3D$.

Contains values of the boundary condition on the leftmost boundary of the domain along the $y$-axis (for more information, refer to a detailed description of $bd_ay$).

$bd_by$  
double* for $d_commit_Helmholtz_2D/d_commit_Helmholtz_3D$,  
float* for $s_commit_Helmholtz_2D/s_commit_Helmholtz_3D$.

Contains values of the boundary condition on the rightmost boundary of the domain along the $y$-axis (for more information, refer to a detailed description of $bd_by$).

$bd_az$  
double* for $d_commit_Helmholtz_3D$,  
float* for $s_commit_Helmholtz_3D$.

Used only by $?_commit_Helmholtz_3D$. Contains values of the boundary condition on the leftmost boundary of the domain along the $z$-axis (for more information, refer to a detailed description of $bd_ay$).
double* for d_commit_Helmholtz_3D,
float* for s_commit_Helmholtz_3D.

Used only by ?_commit_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to a detailed description of bd_bz).

Output Parameters

- **f**: Contains right-hand side of the problem, possibly altered on output.
- **ipar**: Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in ipar.
- **dpar**: Contains double-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in dpar and spar.
- **spar**: Contains single-precision data to be used by Fast Helmholtz Solver. Modified on output as explained in dpar and spar.
- **xhandle, yhandle**: DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). yhandle is used only by ?_commit_Helmholtz_3D.
- **stat**: MKL_INT*. Routine completion status, which is also written to ipar[0]. Continue to call other Poisson Solver routines only if the status is 0.

Description

The ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines check the consistency and correctness of the parameters to be passed to the solver routines ?_Helmholtz_2D/?_Helmholtz_3D. They also initialize the xhandle and yhandle data structures, ipar array, and dpar or spar array, depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see Caveat on Parameter Modifications.

Unlike ?_init_Helmholtz_2D/?_init_Helmholtz_3D, you must call the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routines in your code. Values of ax, bx, ay, by, az, and bz are passed to the routines with the spar/dpar array, and values of nx, ny, nz, and BCtype are passed with the ipar array.

Return Values

- **stat= 1**: The routine completed without errors but with warnings.
- **stat= 0**: The routine successfully completed the task.
- **stat= -100**: The routine stopped because an error in the input data was found, or the data in the dpar, spar, or ipar array was altered by mistake.
stat= -1000
The routine stopped because of an Intel® MKL FFT or TT interface error.

stat= -10000
The routine stopped because the initialization failed to complete or the parameter ipar[0] was altered by mistake.

stat= -99999
The routine failed to complete the task because of a fatal error.

_Helmholtz_2D/?_Helmholtz_3D
Computes the solution of the 2D/3D Helmholtz problem specified by the parameters.

Syntax

void d_Helmholtz_2D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, const double * dpar, MKL_INT * stat);

void s_Helmholtz_2D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, DFTI_DESCRIPTOR_HANDLE * xhandle, MKL_INT * ipar, const float * spar, MKL_INT * stat);

void d_Helmholtz_3D (double * f, const double * bd_ax, const double * bd_bx, const double * bd_ay, const double * bd_by, const double * bd_az, const double * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, const double * dpar, MKL_INT * stat);

void s_Helmholtz_3D (float * f, const float * bd_ax, const float * bd_bx, const float * bd_ay, const float * bd_by, const float * bd_az, const float * bd_bz, DFTI_DESCRIPTOR_HANDLE * xhandle, DFTI_DESCRIPTOR_HANDLE * yhandle, MKL_INT * ipar, const float * spar, MKL_INT * stat);

Include Files

• mkl.h

Input Parameters

f
do\text{double*} for d_Helmholtz_2D/d_Helmholtz_3D,
do\text{float*} for s_Helmholtz_2D/s_Helmholtz_3D.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate \_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the \_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine, at this point results in an incorrect solution.

• 2D problem: the size of the vector is $(nx+1)*(ny+1)$. The value of the modified right-hand side in the mesh point $(i, j)$ is stored in $f[i+j*(nx+1)]$. 

• 3D problem: the size of the vector is \((nx+1)(ny+1)(nz+1)\). The value of the modified right-hand side in the mesh point \((i, j, k)\) is stored in \(f[i+j*(nx+1)+k*(nx+1)(ny+1)]\).

**xhandle, yhandle**

DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). \(yhandle\) is used only by \(?_\text{Helmholtz}_3D\).

**ipar**

MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver (for details, refer to \(ipar\)).

**dpar**

double array of size depending on the dimension of the problem:

- 2D problem: \(5*nx/2+7\)
- 3D problem: \(5*(nx+ny)/2+9\)

Contains double-precision data to be used by Fast Helmholtz Solver (for details, refer to \(dpar\) and \(spar\)).

**spar**

float array of size depending on the dimension of the problem:

- 2D problem: \(5*nx/2+7\)
- 3D problem: \(5*(nx+ny)/2+9\)

Contains single-precision data to be used by Fast Helmholtz Solver (for details, refer to \(dpar\) and \(spar\)).

**bd_ax**

double* for \(d\_\text{Helmholtz}_2D/d\_\text{Helmholtz}_3D\),

float* for \(s\_\text{Helmholtz}_2D/s\_\text{Helmholtz}_3D\).

Contains values of the boundary condition on the leftmost boundary of the domain along the \(x\)-axis (for more information, refer to a detailed description of \(bd\_ax\)).

**bd_bx**

double* for \(d\_\text{Helmholtz}_2D/d\_\text{Helmholtz}_3D\),

float* for \(s\_\text{Helmholtz}_2D/s\_\text{Helmholtz}_3D\).

Contains values of the boundary condition on the rightmost boundary of the domain along the \(x\)-axis (for more information, refer to a detailed description of \(bd\_bx\)).

**bd_ay**

double* for \(d\_\text{Helmholtz}_2D/d\_\text{Helmholtz}_3D\),

float* for \(s\_\text{Helmholtz}_2D/s\_\text{Helmholtz}_3D\).

Contains values of the boundary condition on the leftmost boundary of the domain along the \(y\)-axis for more information, refer to a detailed description of \(bd\_ay\)).

**bd_by**

double* for \(d\_\text{Helmholtz}_2D/d\_\text{Helmholtz}_3D\),

float* for \(s\_\text{Helmholtz}_2D/s\_\text{Helmholtz}_3D\).

Contains values of the boundary condition on the rightmost boundary of the domain along the \(y\)-axis (for more information, refer to a detailed description of \(bd\_by\)).

**bd_az**

double* for \(d\_\text{Helmholtz}_3D\),
float* for s_Helmholtz_3D.
Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the leftmost boundary of the domain along the z-axis (for more information, refer to a detailed description of bd_az).

bd.bz
double* for d_Helmholtz_3D,
float* for s_Helmholtz_3D.
Used only by ?_Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis (for more information, refer to a detailed description of bd.bz).

NOTE
To avoid incorrect computation results, do not change arrays bd_ax, bd_bx, bd_ay, bd_by, bd_az, bd_bz between a call to the ?_commit_Helmholtz_2D/?_commit_Helmholtz_3D routine and a subsequent call to the appropriate ?_Helmholtz_2D/?_Helmholtz_3D routine.

Output Parameters

f
On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

xhandle, yhandle
Data structures used by the Intel® MKL FFT interface. Although the addresses do not change, the structures are modified on output.

ipar
Contains integer data to be used by Fast Helmholtz Solver. Modified on output as explained in ipar.

stat
MKL_INT*. Routine completion status, which is also written to ipar[0]. Continue to call other Poisson Solver routines only if the status is 0.

Description
The ?_Helmholtz_2D/?_Helmholtz_3D routines compute the approximate solution of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to formulas given in the Poisson Solver Implementation section. The f parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of ax, bx, ay, by, az, and bz are passed to the routines with the spar/dpar array, and values of nx, ny, nz, and BCtype are passed with the ipar array.

Return Values

stat= 1
The routine completed without errors but with some warnings.

stat= 0
The routine successfully completed the task.

stat= -2
The routine stopped because division by zero occurred. It usually happens if the data in the dpar or spar array was altered by mistake.
The routine stopped because the sufficient memory was unavailable for the computations.

The routine stopped because an error in the input data was found or the data in the $dpar$, $spar$, or $ipar$ array was altered by mistake.

The routine stopped because of the Intel® MKL FFT or TT interface error.

The routine stopped because the initialization failed to complete or the parameter $ipar[0]$ was altered by mistake.

The routine failed to complete the task because of a fatal error.

**free_Helmholtz_2D/free_Helmholtz_3D**
Releases the memory allocated for the data structures used by the FFT interface.

**Syntax**

```c
void free_Helmholtz_2D(DFTI_DESCRIPTOR_HANDLE* xhandle, MKL_INT* ipar, MKL_INT* stat);
void free_Helmholtz_3D(DFTI_DESCRIPTOR_HANDLE* xhandle, DFTI_DESCRIPTOR_HANDLE* yhandle, MKL_INT* ipar, MKL_INT* stat);
```

**Include Files**
- mkl.h

**Input Parameters**

- **xhandle, yhandle**
  DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structure $yhandle$ is used only by free_Helmholtz_3D.

- **ipar**
  MKL_INT array of size 128. Contains integer data used by Fast Helmholtz Solver (for details, refer to $ipar$).

**Output Parameters**

- **xhandle, yhandle**
  Data structures used by the Intel® MKL FFT interface. Memory allocated for the structures is released on output.

- **ipar**
  Contains integer data used by Fast Helmholtz Solver. On output, the status of the routine call is written to $ipar[0]$.

- **stat**
  MKL_INT*. Routine completion status, which is also written to $ipar[0]$.

**Description**
The free_Helmholtz_2D-free_Helmholtz_3D routine releases the memory used by the $xhandle$ and $yhandle$ structures, which are needed for calling the Intel® MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.
Return Values

\(\text{stat} = 0\)

The routine successfully completed the task.

\(\text{stat} = -1000\)

The routine stopped because of an Intel® MKL FFT or TT interface error.

\(\text{stat} = -99999\)

The routine failed to complete the task because of a fatal error.

Routines for the Spherical Solver

The section describes Poisson Solver routines for the spherical case, their syntax, parameters, and return values. All flavors of the same routine are described together: single- and double-precision and periodic (having names ending in "p") and non-periodic (having names ending in "np"). These Poisson Solver routines also call the Intel® MKL FFT routines (described in section "FFT Functions" in chapter "Fast Fourier Transforms"), which enhance the performance of the Poisson Solver routines.

\(\_\text{init}\_\text{sph}\_\text{p}/\_\text{init}\_\text{sph}\_\text{np}\)

Initializes basic data structures of the periodic and non-periodic Fast Helmholtz Solver on a sphere.

Syntax

```c
void d_init_sph_p (const double * ap, const double * at, const double * bp, const double * bt, const MKL_INT * np, const MKL_INT *nt, const double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat );
void s_init_sph_p (const float * ap, const float * at, const float * bp, const float * bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float * spar, MKL_INT * stat );
void d_init_sph_np (const double * ap, const double * at, const double * bp, const double * bt, const MKL_INT * np, const MKL_INT * nt, const double * q, MKL_INT * ipar, double * dpar, MKL_INT * stat );
void s_init_sph_np (const float * ap, const float * at, const float * bp, const float * bt, const MKL_INT * np, const MKL_INT * nt, const float * q, MKL_INT * ipar, float * spar, MKL_INT * stat );
```

Include Files

- mkl.h

Input Parameters

\(ap\)

double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.

The coordinate (angle) of the leftmost boundary of the domain along the \(\phi\)-axis.

\(bp\)

double* for d_init_sph_p/d_init_sph_np, float* for s_init_sph_p/s_init_sph_np.

The coordinate (angle) of the rightmost boundary of the domain along the \(\phi\)-axis.
at  
double* for d_init_sph_p/d_init_sph_np,  
float* for s_init_sph_p/s_init_sph_np.
The coordinate (angle) of the leftmost boundary of the domain along the \( \theta \)-axis.

bt  
double* for d_init_sph_p/d_init_sph_np,  
float* for s_init_sph_p/s_init_sph_np.
The coordinate (angle) of the rightmost boundary of the domain along the \( \theta \)-axis.

np  
MKL_INT*. The number of mesh intervals along the \( \phi \)-axis. Must be even in the periodic case.

nt  
MKL_INT*. The number of mesh intervals along the \( \theta \)-axis.

q  
double* for d_init_sph_p/d_init_sph_np,  
float* for s_init_sph_p/s_init_sph_np.
The constant Helmholtz coefficient. To solve the Poisson problem, set the value of \( q \) to 0.

Output Parameters

ipar  
MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).

dpar  
double array of size \( 5*nt/2+nt+10 \). Contains double-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

spar  
float array of size \( 5*nt/2+nt+10 \). Contains single-precision data to be used by Fast Helmholtz Solver on a sphere (for details, refer to dpar and spar).

stat  
MKL_INT*. Routine completion status, which is also written to ipar[0]. Continue to call other Poisson Solver routines only if the status is 0.

Description

The \_init_sph_p/\_init_sph_np routines initialize basic data structures for Poisson Solver computations. All routines invoked after a call to a \_init_Helmholtz_2D/\_init_Helmholtz_3D routine use values of the ipar, dpar, and spar array parameters returned by the routine. A detailed description of the array parameters can be found in Common Parameters.

Caution

Data structures initialized and created by periodic flavors of the routine cannot be used by non-periodic flavors of any Poisson Solver routines for Helmholtz Solver on a sphere, and vice versa.

You can skip calls to these routines in your code. However, see Caveat on Parameter Modifications for information on initializing the data structures.
Return Values

\(stat = 0\)  

The routine successfully completed the task. In general, to proceed with computations, the routine should complete with this \(stat\) value.

\(stat = -99999\)  

The routine failed to complete the task because of fatal error.

\(?_\text{commit}\_\text{sph}\_\text{p}/?_\text{commit}\_\text{sph}\_\text{np}\)

Checks consistency and correctness of input data and initializes certain data structures required to solve the periodic/non-periodic Helmholtz problem on a sphere.

Syntax

```c
void d_commit_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_commit_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_commit_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_commit_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar, MKL_INT* stat);
```

Include Files

- mkl.h

Input Parameters

\(f\)

`double*` for `d_commit_sph_p/d_commit_sph_np`, `float*` for `s_commit_sph_p/s_commit_sph_np`.

Contains the right-hand side of the problem packed in a single vector. The size of the vector is \((np+1)\times(nt+1)\) and value of the right-hand side in the mesh point \((i,j)\) is stored in \(f[i+j\times(np+1)]\).

Note that the array \(f\) may be altered by the routine. Save this vector to another memory location if you want to preserve it.

\(ipar\)

`MKL_INT` array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \(ipar\)).

\(dpar\)

`double` array of size \(5\times np/2+nt+10\). Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \(dpar\) and \(spar\)).

\(spar\)

`float` array of size \(5\times np/2+nt+10\). Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \(dpar\) and \(spar\)).
Output Parameters

\( f \)
Contains the right-hand side of the problem, possibly altered on output.

\( ipar \)
Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( ipar \).

\( dpar \)
Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( dpar \) and \( spar \).

\( spar \)
Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( dpar \) and \( spar \).

\( handle_s, handle_c, handle \)
\( \text{DFTI} \_\text{DESCRIPTOR}_\text{HANDLE}^* \). Data structures used by the Intel\textsuperscript{®} MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). \( handle_s \) and \( handle_c \) are used only in \( \_\text{commit}_\text{sph}_p \) and \( handle \) is used only in \( \_\text{commit}_\text{sph}_np \).

\( stat \)
\( \text{MKL} \_\text{INT}^* \). Routine completion status, which is also written to \( ipar[0] \). Continue to call other Poisson Solver routines only if the status is 0.

Description

The \( \_\text{commit}_\text{sph}_p/\_\text{commit}_\text{sph}_np \) routines check consistency and correctness of the parameters to be passed to the solver routines \( \_\text{sph}_p/\_\text{sph}_np \), respectively. They also initialize certain data structures. The routine \( \_\text{commit}_\text{sph}_p \) initializes structures \( handle_s \) and \( handle_c \), and \( \_\text{commit}_\text{sph}_np \) initializes \( handle \). The routines also initialize the \( ipar \) array and \( dpar \) or \( spar \) array, depending upon the routine precision. Refer to Common Parameters to find out which particular array elements the \( \_\text{commit}_\text{sph}_p/\_\text{commit}_\text{sph}_np \) routines initialize and to what values these elements are initialized.

The routines perform only a basic check for correctness and consistency. If you are going to modify parameters of Poisson Solver routines, see Caveat on Parameter Modifications.

Unlike \( \_\text{init}_\text{sph}_p/\_\text{init}_\text{sph}_np \), you must call the \( \_\text{commit}_\text{sph}_p/\_\text{commit}_\text{sph}_np \) routines.

Values of \( np \) and \( nt \) are passed to each of the routines with the \( ipar \) array.

Return Values

\( stat = 1 \)
The routine completed without errors but with warnings.

\( stat = 0 \)
The routine successfully completed the task.

\( stat = -100 \)
The routine stopped because an error in the input data was found or the data in the \( dpar, spar \), or \( ipar \) array was altered by mistake.

\( stat = -1000 \)
The routine stopped because of an Intel\textsuperscript{®} MKL FFT or TT interface error.

\( stat = -10000 \)
The routine stopped because the initialization failed to complete or the parameter \( ipar[0] \) was altered by mistake.
The routine failed to complete the task because of a fatal error.

?_sph_p/?_sph_np
Computes the solution of the spherical Helmholtz problem specified by the parameters.

Syntax

void d_sph_p(double* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_sph_p(float* f, DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, float* spar, MKL_INT* stat);
void d_sph_np(double* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, double* dpar, MKL_INT* stat);
void s_sph_np(float* f, DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, float* spar, MKL_INT* stat);

Include Files

• mkl.h

Input Parameters

\texttt{f}

double* for \texttt{d_sph_p}/\texttt{d_sph_np},
float* for \texttt{s_sph_p}/\texttt{s_sph_np}.

Contains the right-hand side of the problem packed in a single vector and modified by the appropriate \texttt{?_commit_sph_p/\texttt{?_commit_sph_np}} routine. Note that an attempt to substitute the original right-hand side vector, which was passed to the \texttt{?_commit_sph_p/\texttt{?_commit_sph_np}} routine, at this point results in an incorrect solution.

The size of the vector is \((np+1)*(nt+1)\) and the value of the modified right-hand side in the mesh point \((i, j)\) is stored in \(f[i+j*(np+1)]\).

\texttt{handle_s}, \texttt{handle_c}, \texttt{handle}

DFTI_DESCRIPTOR_HANDLE*. Data structures used by Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). \texttt{handle_s} and \texttt{handle_c} are used only in \texttt{?_sph_p} and \texttt{handle} is used only in \texttt{?_sph_np}.

\texttt{ipar}

MKL_INT array of size 128. Contains integer data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \texttt{ipar}).

\texttt{dpar}

double array of size \(5*nt/2+nt+10\). Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \texttt{dpar} and \texttt{spar}).

\texttt{spar}

float array of size \(5*nt/2+nt+10\). Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere (for details, refer to \texttt{dpar} and \texttt{spar}).
Output Parameters

\( f \)  
On output, contains the approximate solution to the problem packed the same way as the right-hand side of the problem was packed on input.

\( \text{handle}_s, \text{handle}_c, \text{handle} \)  
Data structures used by the Intel® MKL FFT interface.

\( \text{ipar} \)  
Contains integer data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( \text{ipar} \).

\( \text{dpar} \)  
Contains double-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( \text{dpar} \) and \( \text{spar} \).

\( \text{spar} \)  
Contains single-precision data to be used by the Fast Helmholtz Solver on a sphere. Modified on output as explained in \( \text{dpar} \) and \( \text{spar} \).

\( \text{stat} \)  
MKL_INT*. Routine completion status, which is also written to \( \text{ipar}[0] \). Continue to call other Poisson Solver routines only if the status is 0.

Description

The sph_p/sph_np routines compute the approximate solution on a sphere of the Helmholtz problem defined in the previous calls to the corresponding initialization and commit routines. The solution is computed according to the formulas given in the Poisson Solver Implementation section. The \( f \) parameter, which initially holds the packed vector of the right-hand side of the problem, is replaced by the computed solution packed in the same way. Values of \( np \) and \( nt \) are passed to each of the routines with the \( \text{ipar} \) array.

Return Values

\( \text{stat} = 1 \)  
The routine completed without errors but with warnings.

\( \text{stat} = 0 \)  
The routine successfully completed the task.

\( \text{stat} = -2 \)  
The routine stopped because division by zero occurred. It usually happens if the data in the \( \text{dpar} \) or \( \text{spar} \) array was altered by mistake.

\( \text{stat} = -3 \)  
The routine stopped because the memory was insufficient to complete the computations.

\( \text{stat} = -100 \)  
The routine stopped because an error in the input data was found or the data in the \( \text{dpar}, \text{spar}, \) or \( \text{ipar} \) array was altered by mistake.

\( \text{stat} = -1000 \)  
The routine stopped because of an Intel® MKL FFT or TT interface error.

\( \text{stat} = -10000 \)  
The routine stopped because the initialization failed to complete or the parameter \( \text{ipar}[0] \) was altered by mistake.

\( \text{stat} = -99999 \)  
The routine failed to complete the task because of a fatal error.
free_sph_p/free_sph_np
Releases the memory allocated for the data structures
used by the FFT interface.

Syntax
void free_sph_p(DFTI_DESCRIPTOR_HANDLE* handle_s, DFTI_DESCRIPTOR_HANDLE* handle_c, MKL_INT* ipar, MKL_INT* stat);
void free_sph_np(DFTI_DESCRIPTOR_HANDLE* handle, MKL_INT* ipar, MKL_INT* stat);

Include Files
• mkl.h

Input Parameters
handle_s, handle_c, handle
DFTI_DESCRIPTOR_HANDLE*. Data structures used by the Intel® MKL FFT interface (for details, refer to section "FFT Functions" in chapter "Fast Fourier Transforms"). The structures handle_s and handle_c are used only in free_sph_p, and handle is used only in free_sph_np.

ipar
MKL_INT array of size 128. Contains integer data to be used by Fast Helmholtz Solver on a sphere (for details, refer to ipar).

Output Parameters
handle_s, handle_c, handle
Data structures used by the Intel® MKL FFT interface. Memory allocated for the structures is released on output.

ipar
Contains integer data to be used by Fast Helmholtz Solver on a sphere. On output, the status of the routine call is written to ipar[0].

stat
MKL_INT*. Routine completion status, which is also written to ipar[0].

Description
The free_sph_p/free_sph_np routine releases the memory used by the handle_s, handle_c or handle structures, needed for calling the Intel® MKL FFT functions. To release memory allocated for other parameters, include memory release statements in your code.

Return Values

stat= 0
The routine successfully completed the task.

stat=-1000
The routine stopped because of an Intel® MKL FFT or TT interface error.

stat=-99999
The routine failed to complete the task because of a fatal error.

Common Parameters for the Poisson Solver
ipar

MKL_INT array of size 128, holds integer data needed for Fast Helmholtz Solver (both for Cartesian and spherical coordinate systems). Its elements are described in Table "Elements of the ipar Array":

NOTE
Initial values are assigned to the array parameters by the appropriate `_init_Helmholtz_2D/_init_Helmholtz_3D/_init_sph_p/_init_sph_np and `_commit_Helmholtz_2D/_commit_Helmholtz_3D/_commit_sph_p/_commit_sph_np routines.

## Elements of the ipar Array

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0     | Contains status value of the last Poisson Solver routine called. In general, it should be 0 on exit from a routine to proceed with the Fast Helmholtz Solver. The element has no predefined values. This element can also be used to inform the `_commit_Helmholtz_2D/_commit_Helmholtz_3D/_commit_sph_p/_commit_sph_np routines of how the Commit step of the computation should be carried out (see Figure "Typical Order of Invoking Poisson Solver Routines"). A non-zero value of ipar[0] with decimal representation $abc = 100a + 10b + c$

$=100a+10b+c$, where each of $a$, $b$, and $c$ is equal to 0 or 9, indicates that some parts of the Commit step should be omitted.

- If $c=9$, the routine omits checking of parameters and initialization of the data structures.
- If $b=9$,
  - In the Cartesian case, the routine omits the adjustment of the right-hand side vector $f$ to the Neumann boundary condition (multiplication of boundary values by 0.5 as well as incorporation of the boundary function $g$) and/or the Dirichlet boundary condition (setting boundary values to 0 as well as incorporation of the boundary function $G$).
  - For the Helmholtz solver on a sphere, the routine omits computation of the spherical weights for the dpar/spar array.
- If $a=9$, the routine omits the normalization of the right-hand side vector $f$. Depending on the solver, the normalization means:
  - 2D Cartesian case: multiplication by $h_y^2$, where $h_y$ is the mesh size in the $y$ direction (for details, see Poisson Solver Implementation).
  - 3D (Cartesian) case: multiplication by $h_z^2$, where $h_z$ is the mesh size in the $z$ direction.
  - Helmholtz solver on a sphere: multiplication by $h_\theta^2$, where $h_\theta$ is the mesh size in the $\theta$ direction (for details, see Poisson Solver Implementation).

Using ipar[0] you can adjust the routine to your needs and improve efficiency in solving multiple Helmholtz problems that differ only in the right-hand side. You must be cautious when using this method, because any misunderstanding of the commit process may cause incorrect results or program failure (see also Caveat on Parameter Modifications).

1
Contains error messaging options:

- ipar[1]=-1 indicates that all error messages are printed to the MKL_Poisson_Library_log.txt file in the folder from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device (usually, screen).
- ipar[1]=0 indicates that no error messages will be printed.
- ipar[1]=1 is the default value. It indicates that all error messages are printed to the standard output device.
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2     | Contains warning messaging options:  
- ipar[2]=-1 indicates that all warning messages are printed to the MKL_Poisson_Library_log.txt file in the directory from which the routine is called. If the file does not exist, the routine tries to create it. If the attempt fails, the routine prints information that the file cannot be created to the standard output device.  
- ipar[2]=0 indicates that no warning messages will be printed.  
- ipar[2]=1 is the default value. It indicates that all warning messages are printed to the standard output device.  
In case of warnings, the stat parameter contains a non-zero value on exit from a routine regardless of the ipar[2] setting. |
| 3     | Unused. Parameters 4 through 9 are used only in the Cartesian case. |
| 4     | Takes this value:  
- 2, if BCtype[0] ='P'  
- 1, if BCtype[0] ='N'  
- 0, if BCtype[0] ='D'  
- -1, otherwise |
| 5     | Takes this value:  
- 2, if BCtype[1] ='P'  
- 1, if BCtype[1] ='N'  
- 0, if BCtype[1] ='D'  
- -1, otherwise |
| 6     | Takes this value:  
- 2, if BCtype[2] ='P'  
- 1, if BCtype[2] ='N'  
- 0, if BCtype[2] ='D'  
- -1, otherwise |
| 7     | Takes this value:  
- 2, if BCtype[3] ='P'  
- 1, if BCtype[3] ='N'  
- 0, if BCtype[3] ='D'  
- -1, otherwise |
| 8     | Takes this value:  
- 2, if BCtype[4] ='P'  
- 1, if BCtype[4] ='N'  
- 0, if BCtype[4] ='D'  
- -1, otherwise |
| 9     | Takes this value:  
- 2, if BCtype[5] ='P'  
- 1, if BCtype[5] ='N'  
- 0, if BCtype[5] ='D'  
- -1, otherwise |
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
</table>
| 10    | Takes the value of  
|       | • $nx$, that is, the number of intervals along the $x$-axis, in the Cartesian case.  
|       | • $np$, that is, the number of intervals along the $\varphi$-axis, in the spherical case.  |
| 11    | Takes the value of  
|       | • $ny$, that is, the number of intervals along the $y$-axis, in the Cartesian case  
|       | • $nt$, that is, the number of intervals along the $\theta$-axis, in the spherical case.  |
| 12    | Takes the value of $nz$, the number of intervals along the $z$-axis. This parameter is used only in the 3D case (Cartesian).  |
| 13    | Has the value of 6, which specifies the internal partitioning of the $dpar/spar$ array.  |

The values of $ipar[15] - ipar[20]$ depend on the dimension of the problem for the Cartesian solver or on whether the solver on a sphere is periodic.

<table>
<thead>
<tr>
<th>2D case</th>
<th>3D case</th>
<th>Periodic case</th>
<th>Non-periodic case</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>Unused</td>
<td>Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Takes the value of $ipar[14]+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[16]+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>Takes the value of $ipar[14]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[16]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[16]+3*ipar[10]/2+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
</tr>
<tr>
<td>19</td>
<td>Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[18]+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Unused</td>
</tr>
<tr>
<td>20</td>
<td>Takes the value of $ipar[19]+3*ipar[12]/4$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[18]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
<td>Takes the value of $ipar[18]+3*ipar[10]/4+1$, which specifies the internal partitioning of the $dpar/spar$ array.</td>
</tr>
</tbody>
</table>

The values of $ipar[21] - ipar[119]$ are assigned regardless of the dimension of the problem for the Cartesian solver or of whether the solver on a sphere is periodic.
### Index Description

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
</table>
| 21    | Contains message style options. Specifically:  
• `ipar[21]=1` (default) indicates that Poisson Solver routines print the messages in C-style notations. |
| 22    | Contains the number of OpenMP threads to be used for computations in a multithreaded environment. The default value is 1 in the serial mode, and the result returned by the `mkl_get_max_threads` function otherwise. |
| 23    | Takes the value of `ipar[18]+1`, which specifies the internal partitioning of the `dpar/spar` array in the periodic Cartesian case. |
| 24    | Takes the value of `ipar[23]+3*ipar[12]/4`, which specifies the internal partitioning of the `dpar/spar` array in the periodic Cartesian case. |
| 25    | Takes the value of `ipar[20]+1`, which specifies the internal partitioning of the `dpar/spar` array in the periodic 3D Cartesian case. |
| 26    | Takes the value of `ipar[25]+3*ipar[13]/4`, which specifies the internal partitioning of the `dpar/spar` array in the periodic 3D Cartesian case. |
| 27 through 39 | Unused. |
| 40 through 59 | Contain the first twenty elements of the `ipar` array of the first Trigonometric Transform that the solver uses. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.) |
| 60 through 79 | Contain the first twenty elements of the `ipar` array of the second Trigonometric Transform that the 3D Cartesian and periodic spherical solvers use. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.) |
| 80 through 99 | Contain the first twenty elements of the `ipar` array of the third Trigonometric Transform that the solver uses in case of periodic boundary conditions along the x-axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.) |
| 100 through 119 | Contain the first twenty elements of the `ipar` array of the fourth Trigonometric Transform used by periodic spherical solvers and 3D Cartesian solvers with periodic boundary conditions along the y-axis. (For details, see Common Parameters in the "Trigonometric Transform Routines" section.) |

**NOTE**  
While you can declare the `ipar` array as `MKL_INT ipar[120]`, for future compatibility you should declare `ipar` as `MKL_INT ipar[128]`.  

### `dpar` and `spar`  
Arrays `dpar` and `spar` are the same except in the data precision:  

**dpar**  
Holds data needed for double-precision Fast Helmholtz Solver computations.  
• For the Cartesian solver, double array of size `5*nx/2+7` in the 2D case or `5*(nx+ny)/2+9` in the 3D case; initialized in the `d_init_Helmholtz_2D/d_init_Helmholtz_3D` and `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` routines.
• For the spherical solver, double array of size 5*\(np/2 + nt + 10\); initialized in the 
  \texttt{d\_init\_sph\_p/d\_init\_sph\_np} and \texttt{d\_commit\_sph\_p/d\_commit\_sph\_np} 
  routines.

\textit{spar}

Holds data needed for single-precision Fast Helmholtz Solver computations.

• For the Cartesian solver, float array of size 5*\(nx/2 + 7\) in the 2D case or 
  5*(\(nx+ny\))/2+9 in the 3D case; initialized in the 
  \texttt{s\_init\_Helmholtz\_2D/s\_init\_Helmholtz\_3D} and 
  \texttt{s\_commit\_Helmholtz\_2D/s\_commit\_Helmholtz\_3D} routines.

• For the spherical solver, float array of size 5*\(np/2 + nt + 10\); initialized in the 
  \texttt{s\_init\_sph\_p/s\_init\_sph\_np} and \texttt{s\_commit\_sph\_p/s\_commit\_sph\_np} 
  routines.

Because \texttt{dpar} and \texttt{spar} have similar elements in each position, the elements are described together in Table "Elements of the \texttt{dpar} and \texttt{spar} Arrays":

\textbf{Elements of the \texttt{dpar} and \texttt{spar} Arrays}

<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0     | In the Cartesian case, contains the length of the interval along the \(x\)-axis right after a call to the \texttt{?\_init\_Helmholtz\_2D/?\_init\_Helmholtz\_3D} routine or the mesh size \(h_x\) in the \(x\) direction (for details, see \textit{Poisson Solver Implementation}) after a call to the \texttt{?\_commit\_Helmholtz\_2D/?\_commit\_Helmholtz\_3D} routine. 

In the spherical case, contains the length of the interval along the \(\phi\)-axis right after a call to the \texttt{?\_init\_sph\_p/?\_init\_sph\_np} routine or the mesh size \(h_{\phi}\) in the \(\phi\) direction (for details, see \textit{Poisson Solver Implementation}) after a call to the \texttt{?\_commit\_sph\_p/?\_commit\_sph\_np} routine. |
| 1     | In the Cartesian case, contains the length of the interval along the \(y\)-axis right after a call to the \texttt{?\_init\_Helmholtz\_2D/?\_init\_Helmholtz\_3D} routine or the mesh size \(h_y\) in the \(y\) direction (for details, see \textit{Poisson Solver Implementation}) after a call to the \texttt{?\_commit\_Helmholtz\_2D/?\_commit\_Helmholtz\_3D} routine. 

In the spherical case, contains the length of the interval along the \(\theta\)-axis right after a call to the \texttt{?\_init\_sph\_p/?\_init\_sph\_np} routine or the mesh size \(h_{\theta}\) in the \(\theta\) direction (for details, see \textit{Poisson Solver Implementation}) after a call to the \texttt{?\_commit\_sph\_p/?\_commit\_sph\_np} routine. |
| 2     | In the Cartesian case, contains the length of the interval along the \(z\)-axis right after a call to the \texttt{?\_init\_Helmholtz\_2D/?\_init\_Helmholtz\_3D} routine or the mesh size \(h_z\) in the \(z\) direction (for details, see \textit{Poisson Solver Implementation}) after a call to the \texttt{?\_commit\_Helmholtz\_2D/?\_commit\_Helmholtz\_3D} routine. 

In the spherical solver, this parameter is used only in the 3D case. 

In the spherical solver, contains the coordinate of the leftmost boundary along the \(\theta\)-axis after a call to the \texttt{?\_init\_sph\_p/?\_init\_sph\_np} routine. |
<p>| 3     | Contains the value of the coefficient (q) after a call to the \texttt{?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np} routine. |
| 4     | Contains the tolerance parameter after a call to the \texttt{?_init_Helmholtz_2D/?_init_Helmholtz_3D/?_init_sph_p/?_init_sph_np} routine. |</p>
<table>
<thead>
<tr>
<th>Index</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>In the Cartesian case, this value is used only for the pure Neumann boundary conditions. This is a special case, because the right-hand side of the problem cannot be arbitrary if the coefficient ( q ) is zero. The Poisson Solver verifies that the classical solution exists (up to rounding errors) using this tolerance. In any case, the Poisson Solver computes the normal solution, that is, the solution that has the minimal Euclidean norm of residual. Nevertheless, the <code>_Helmholtz_2D/_Helmholtz_3D</code> routine informs you that the solution may not exist in a classical sense (up to rounding errors).</td>
</tr>
<tr>
<td>•</td>
<td>In the spherical case, the value is used for the special case of a periodic problem on the entire sphere. This special case is similar to the Cartesian case with pure Neumann boundary conditions. Here the Poisson Solver computes the normal solution as well. The parameter is also used to detect whether the problem is periodic up to rounding errors.</td>
</tr>
</tbody>
</table>

The default value for this parameter is 1.0E-10 in case of double-precision computations or 1.0E-4 in case of single-precision computations. You can increase the value of the tolerance, for instance, to avoid the warnings that may appear.

- `ipar[13]-1` through `ipar[14]-1`  
  In the Cartesian case, contain the spectrum of the one-dimensional (1D) problem along the \( x \)-axis after a call to the `_commit_Helmholtz_2D/_commit_Helmholtz_3D` routine.

- `ipar[15]-1` through `ipar[16]-1`  
  In the spherical case, contains the spectrum of the 1D problem along the \( \varphi \)-axis after a call to the `_commit_sph_p/_commit_sph_np` routine.

- `ipar[17]-1` through `ipar[18]-1`  
  Take the values of the (staggered) sine/cosine in the mesh points:
  - along the \( x \)-axis after a call to the `_commit_Helmholtz_2D/_commit_Helmholtz_3D` routine for a Cartesian solver
  - along the \( \varphi \)-axis after a call to the `_commit_sph_p/_commit_sph_np` routine for a spherical solver.

- `ipar[19]-1` through `ipar[20]-1`  
  Take the values of the (staggered) sine/cosine in the mesh points:
  - along the \( y \)-axis after a call to the `_commit_Helmholtz_3D` routine for a Cartesian 3D solver
  - along the \( \varphi \)-axis after a call to the `_commit_sph_p` routine for a spherical periodic solver.

  These elements are not used in the 2D Cartesian case and in the non-periodic spherical case.

- `ipar[23]-1` through `ipar[24]-1`  
  Take the values of the (staggered) sine/cosine in the mesh points along the \( x \)-axis after a call to the `_commit_Helmholtz_2D/_commit_Helmholtz_3D` routine. These elements are used only in the periodic Cartesian case.

- `ipar[25]-1` through `ipar[26]-1`  
  Take the values of the (staggered) sine/cosine in the mesh points along the \( x \)-axis after a call to the `_commit_Helmholtz_3D` routine. These elements are used only in the periodic 3D Cartesian case.
NOTE
You may define the array size depending upon the type of the problem to solve.

Caveat on Parameter Modifications

Flexibility of the Poisson Solver interface enables you to skip calls to the \texttt{? init Helmholtz 2D/? init Helmholtz 3D/? init sph p/? init sph np} routine and to initialize the basic data structures explicitly in your code. You may also need to modify contents of the \texttt{ipar}, \texttt{dpar}, and \texttt{spar} arrays after initialization. When doing so, provide correct and consistent data in the arrays. Mistakenly altered arrays cause errors or incorrect results. You can perform a basic check for correctness and consistency of parameters by calling the \texttt{? commit Helmholtz 2D/? commit Helmholtz 3D} routine; however, this does not ensure the correct solution but only reduces the chance of errors or wrong results.

NOTE
To supply correct and consistent parameters to Poisson Solver routines, you should have considerable experience in using the Poisson Solver interface and good understanding of the solution process, as well as elements contained in the \texttt{ipar}, \texttt{spar}, and \texttt{dpar} arrays and dependencies between values of these elements.

In rare occurrences when you fail in tuning parameters for the Fast Helmholtz Solver, refer for technical support at http://www.intel.com/software/products/support/.

WARNING
The only way that ensures a proper solution of a Helmholtz problem is to follow a typical sequence of invoking the routines and not change the default set of parameters. So, avoid modifications of \texttt{ipar}, \texttt{dpar}, and \texttt{spar} arrays unless it is necessary.

Parameters That Define Boundary Conditions

Poisson Solver routines for the Cartesian solver use the following common parameters to define the boundary conditions.

Parameters to Define Boundary Conditions for the Cartesian Solver

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{bd_ax}</td>
<td>double* for \texttt{d_commit Helmholtz 2D/d_commit Helmholtz 3D} and \texttt{d Helmholtz 2D/d Helmholtz 3D}, float* for \texttt{s_commit Helmholtz 2D/s_commit Helmholtz 3D} and \texttt{s Helmholtz 2D/s Helmholtz 3D}. Contains values of the boundary condition on the leftmost boundary of the domain along the x-axis.</td>
</tr>
</tbody>
</table>

- 2D problem: the size of the array is \texttt{ny}+1. Its contents depend on the boundary conditions as follows:
  - Dirichlet boundary condition (value of \texttt{BCTYPE[0]} is ‘D’): values of the function \(G(ax, y_j), j=0, \ldots, ny.\)
  - Neumann boundary condition (value of \texttt{BCTYPE[0]} is ‘N’): values of the function \(g(ax, y_j), j=0, \ldots, ny.\)

  The value corresponding to the index \(j\) is placed in \texttt{bd\_ax}[j].

- 3D problem: the size of the array is \(ny+1)*(nz+1). Its contents depend on the boundary conditions as follows:
### Parameter Description

- **Dirichlet boundary condition** (value of `BCtype[0]` is 'D'): values of the function \( G(ax, y_j, z_k), j=0, \ldots, ny, k=0, \ldots, nz. \)
- **Neumann boundary condition** (value of `BCtype[0]` is 'N'): the values of the function \( g(ax, y_j, z_k), j=0, \ldots, ny, k=0, \ldots, nz. \)

The values are packed in the array so that the value corresponding to indices \((j, k)\) is placed in \( bd\_ax[j+k*(ny+1)] \).

For periodic boundary conditions (the value of `BCtype[0]` is 'P'), this parameter is not used, so it can accept a dummy pointer.

**bd\_bx**

- **2D problem**: the size of the array is \( ny+1 \). Its contents depend on the boundary conditions as follows:
  - Dirichlet boundary condition (value of `BCtype[1]` is 'D'): values of the function \( G(bx, y_j), j=0, \ldots, ny. \)
  - Neumann boundary condition (value of `BCtype[1]` is 'N'): values of the function \( g(bx, y_j), j=0, \ldots, ny. \)

The value corresponding to the index \( j \) is placed in \( bd\_bx[j] \).

- **3D problem**: the size of the array is \( (ny+1)*(nz+1) \). Its contents depend on the boundary conditions as follows:
  - Dirichlet boundary condition (value of `BCtype[1]` is 'D'): values of the function \( G(bx, y_j, z_k), j=0, \ldots, ny, k=0, \ldots, nz. \)
  - Neumann boundary condition (value of `BCtype[1]` is 'N'): values of the function \( g(bx, y_j, z_k), j=0, \ldots, ny, k=0, \ldots, nz. \)

The values are packed in the array so that the value corresponding to indices \((j, k)\) is placed in \( bd\_bx[j+k*(ny+1)] \).

For periodic boundary conditions (the value of `BCtype[1]` is 'P'), this parameter is not used, so it can accept a dummy pointer.

**bd\_ay**

- **2D problem**: the size of the array is \( nx+1 \). Its contents depend on the boundary conditions as follows:
  - Dirichlet boundary condition (value of `BCtype[2]` is 'D'): values of the function \( G(x_i, ay), i=0, \ldots, nx. \)
  - Neumann boundary condition (value of `BCtype[2]` is 'N'): values of the function \( g(x_i, ay), i=0, \ldots, nx. \)

The value corresponding to the index \( i \) is placed in \( bd\_ay[i] \).

- **3D problem**: the size of the array is \( (nx+1)*(nz+1) \). Its contents depend on the boundary conditions as follows:
Parameter | Description
---|---
- Dirichlet boundary condition (value of `BCtype[2]` is 'D'): values of the function $G(x_i, ay, z_k), i=0, \ldots, nx, k=0, \ldots, nz$.  
- Neumann boundary condition (value of `BCtype[2]` is 'N'): values of the function $g(x_i, ay, z_k), i=0, \ldots, nx, k=0, \ldots, nz$.

The values are packed in the array so that the value corresponding to indices $(i, k)$ is placed in $bd_ay[i+k*(nx+1)]$.  

For periodic boundary conditions (the value of `BCtype[2]` is 'P'), this parameter is not used, so it can accept a dummy pointer.

$bd_by$  
double* for `d_commit_Helmholtz_2D/d_commit_Helmholtz_3D` and `d_Helmholtz_2D/d_Helmholtz_3D`,  
float* for `s_commit_Helmholtz_2D/s_commit_Helmholtz_3D` and `s_Helmholtz_2D/s_Helmholtz_3D`.  
Contains values of the boundary condition on the rightmost boundary of the domain along the $y$-axis.

- 2D problem: the size of the array is $nx+1$. Its contents depend on the boundary conditions as follows:  
  - Dirichlet boundary condition (value of `BCtype[3]` is 'D'): values of the function $G(x_i, by), i=0, \ldots, nx$.  
  - Neumann boundary condition (value of `BCtype[3]` is 'N'): values of the function $g(x_i, by), i=0, \ldots, nx$.

The value corresponding to the index $i$ is placed in $bd_by[i]$.

- 3D problem: the size of the array is $(nx+1)*(nz+1)$. Its contents depend on the boundary conditions as follows:  
  - Dirichlet boundary condition (value of `BCtype[3]` is 'D'): values of the function $G(x_i, by, z_k), i=0, \ldots, nx, k=0, \ldots, nz$.  
  - Neumann boundary condition (value of `BCtype[3]` is 'N'): values of the function $g(x_i, by, z_k), i=0, \ldots, nx, k=0, \ldots, nz$.

The values are packed in the array so that the value corresponding to indices $(i, k)$ is placed in $bd_by[i+k*(nx+1)]$.

For periodic boundary conditions (the value of `BCtype[3]` is 'P'), this parameter is not used, so it can accept a dummy pointer.

$bd_az$  
double* for `d_commit_Helmholtz_3D` and `d_Helmholtz_3D`,  
float* for `s_commit_Helmholtz_3D` and `s_Helmholtz_3D`.  
Used only by `?_commit_Helmholtz_3D` and `?_Helmholtz_3D`. Contains values of the boundary condition on the leftmost boundary of the domain along the $z$-axis.

The size of the array is $(nx+1)*(ny+1)$. Its contents depend on the boundary conditions as follows:  
- Dirichlet boundary condition (value of `BCtype[4]` is 'D'): values of the function $G(x_i, y_j, az), i=0, \ldots, nx, j=0, \ldots, ny$.  
- Neumann boundary condition (value of `BCtype[4]` is 'N'), values of the function $g(x_i, y_j, az), i=0, \ldots, nx, j=0, \ldots, ny$.

The values are packed in the array so that the value corresponding to indices $(i, j)$ is placed in $bd_az[i+j*(nx+1)]$.

For periodic boundary conditions (the value of `BCtype[4]` is 'P'), this parameter is not used, so it can accept a dummy pointer.
Parameter | Description
---|---
bd_bz | double* for d_commit_Helmholtz_3D and d_Helmholtz_3D, float* for s_commit_Helmholtz_3D and s_Helmholtz_3D.

Used only by _commit_Helmholtz_3D and _Helmholtz_3D. Contains values of the boundary condition on the rightmost boundary of the domain along the z-axis.

The size of the array is \((nx+1)\times(ny+1)\). Its contents depend on the boundary conditions as follows:

- Dirichlet boundary condition (value of BCtype[5] is 'D'): values of the function \(G(x_i, y_j, bz)\), \(i=0, \ldots, nx, j=0, \ldots, ny\).
- Neumann boundary condition (value of BCtype[5] is 'N'): values of the function \(g(x_i, y_j, bz)\), \(i=0, \ldots, nx, j=0, \ldots, ny\).

The values are packed in the array so that the value corresponding to indices \((i, j)\) is placed in \(bd_bz[i+j\times(nx+1)]\).

For periodic boundary conditions (the value of BCtype[5] is 'P'), this parameter is not used, so it can accept a dummy pointer.

See Also

_commit_Helmholtz_2D/?_commit_Helmholtz_3D
_Helmholtz_2D/?_Helmholtz_3D

Poisson Solver Implementation Details

Several aspects of the Intel® MKL Poisson Solver interface are platform-specific and language-specific. To promote portability of the Intel® MKL Poisson Solver interface across platforms and ease of use across different languages, Intel® MKL provides you with the Poisson Solver language-specific header file to include in your code:

- mkl_poisson.h, to be used together with mkl_dfti.h.

**NOTE**

- Use of the Intel® MKL Poisson Solver software without including the above language-specific header files is not supported.

Header File

The header file defines the function prototypes for the Cartesian and spherical solver available in specific function descriptions.
Nonlinear Optimization Problem Solvers

Intel® Math Kernel Library (Intel® MKL) provides tools for solving nonlinear least squares problems using the Trust-Region (TR) algorithms. The solver routines are grouped according to their purpose as follows:

- Nonlinear Least Squares Problem without Constraints
- Nonlinear Least Squares Problem with Linear (Boundary) Constraints
- Jacobian Matrix Calculation Routines

For more information on the key concepts required to understand the use of the Intel® MKL nonlinear least squares problem solver routines, see [Conn00].

Nonlinear Solver Organization and Implementation

The Intel® MKL solver routines for nonlinear least squares problems use reverse communication interfaces (RCI). That means you need to provide the solver with information required for the iteration process, for example, the corresponding Jacobian matrix, or values of the objective function. RCI removes the dependency of the solver on specific implementation of the operations. However, it does require that you organize a computational loop.

Typical order for invoking RCI solver routines

The nonlinear least squares problem solver routines, or Trust-Region (TR) solvers, are implemented with threading support. You can manage the threads using Threading Control Functions. The TR solvers use BLAS and LAPACK routines, and offer the same parallelism as those domains. The ?jacobi and ?jacobix routines of Jacobi matrix calculations are parallel. These routines (?jacobi and ?jacobix) make calls to the user-supplied functions with different x parameters for multiple threads.
Memory Allocation and Handles

To make the TR solver routines easy to use, you are not required to allocate temporary working storage. The solver allocates any required memory. To allow multiple users to access the solver simultaneously, the solver keeps track of the storage allocated for a particular application by using a data object called a handle. Each TR solver routine creates, uses, or deletes a handle. To declare a handle, include `mkl.h`.

Declare a handle as one of the following:

```c
#include "mkl.h"
_TRNSP_HANDLE_t handle;
```

or

```c
_TRNSPBC_HANDLE_t handle;
```

The first declaration is used for nonlinear least squares problems without boundary constraints, and the second is used for nonlinear least squares problems with boundary constraints.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Nonlinear Solver Routine Naming Conventions

The TR routine names have the following structure:

```
<character><name>_<action>( )
```

where

- `<character>` indicates the data type:
  - `s` float
  - `d` double

- `<name>` indicates the task type:
  - `trnlsp` nonlinear least squares problem without constraints
  - `trnlspbc` nonlinear least squares problem with boundary constraints
  - `jacobi` computation of the Jacobian matrix using central differences

- `<action>` indicates an action on the task:
  - `init` initializes the solver
  - `check` checks correctness of the input parameters
  - `solve` solves the problem
  - `get` retrieves the number of iterations, the stop criterion, the initial residual, and the final residual
delete releases the allocated data

Nonlinear Least Squares Problem without Constraints

The nonlinear least squares problem without constraints can be described as follows:

$$\min_{x \in \mathbb{R}^n} \| F(x) \|_2 = \min_{x \in \mathbb{R}^n} \| y - f(x) \|_2^2, y \in \mathbb{R}^m, x \in \mathbb{R}^n, f: \mathbb{R}^n \rightarrow \mathbb{R}^m, m \geq n,$$

where

$$F(x): \mathbb{R}^n \rightarrow \mathbb{R}^m$$

is a twice differentiable function in $$\mathbb{R}^n$$.

Solving a nonlinear least squares problem means searching for the best approximation to the vector $$y$$ with the model function $$f_i(x)$$ and nonlinear variables $$x$$. The best approximation means that the sum of squares of residuals $$y_i - f_i(x)$$ is the minimum.

See usage examples in the examples\solverc\source folder of your Intel® MKL directory. Specifically, see ex_nlsqp_f.fex_nlsqp_c.c.

### RCI TR Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>?trnlsp_init</td>
<td>Initializes the solver.</td>
</tr>
<tr>
<td>?trnlsp_check</td>
<td>Checks correctness of the input parameters.</td>
</tr>
<tr>
<td>?trnlsp_solve</td>
<td>Solves a nonlinear least squares problem using the Trust-Region algorithm.</td>
</tr>
<tr>
<td>?trnlsp_get</td>
<td>Retrieves the number of iterations, stop criterion, initial residual, and final residual.</td>
</tr>
<tr>
<td>?trnlsp_delete</td>
<td>Releases allocated data.</td>
</tr>
</tbody>
</table>

### ?trnlsp_init

*Initializes the solver of a nonlinear least squares problem.*

**Syntax**

```c
MKL_INT strnlsp_init (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* x, const float* eps, const MKL_INT* iter1, const MKL_INT* iter2, const float* rs);
```

```c
MKL_INT dtrnlsp_init (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* x, const double* eps, const MKL_INT* iter1, const MKL_INT* iter2, const double* rs);
```

**Include Files**

- mkl.h

**Description**

The ?trnlsp_init routine initializes the solver.

After initialization, all subsequent invocations of the ?trnlsp_solve routine should use the values of the handle returned by ?trnlsp_init. This handle stores internal data, including pointers to the arrays $$x$$ and $$eps$$. It is important to not move or deallocate these arrays until after calling the ?trnlsp_delete routine.
The \( \text{eps} \) array contains a number indicating the stopping criteria:

<table>
<thead>
<tr>
<th>( \text{eps} ) Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( \Delta &lt; \text{eps}[0] )</td>
</tr>
<tr>
<td>1</td>
<td>( ||F(x)||_2 &lt; \text{eps}[1] )</td>
</tr>
<tr>
<td>2</td>
<td>The Jacobian matrix is singular. ( ||J(x)_{m\times(j-1)\ldots m\times j-1}||_2 &lt; \text{eps}[2] ), ( j = 1, \ldots, n )</td>
</tr>
<tr>
<td>3</td>
<td>( |s|_2 &lt; \text{eps}[3] )</td>
</tr>
<tr>
<td>4</td>
<td>( ||F(x)||_2 - ||F(x) - J(x)s||_2 &lt; \text{eps}[4] )</td>
</tr>
<tr>
<td>5</td>
<td>The trial step precision. If ( \text{eps}[5] = 0 ), then the trial step meets the required precision (( \leq 1.0\times10^{-10} )).</td>
</tr>
</tbody>
</table>

Note:
- \( J(x) \) is the Jacobian matrix.
- \( \Delta \) is the trust-region area.
- \( F(x) \) is the value of the functional.
- \( s \) is the trial step.

**Input Parameters**

\( n \)

Length of \( x \).

\( m \)

Length of \( F(x) \).

\( x \)

Array of size \( n \). Initial guess. A reference to this array is stored in \( \text{handle} \) for later use and modification by \( \text{trnlsp_solve} \).

\( \text{eps} \)

Array of size 6; contains stopping criteria. See the values in the Description section.

A reference to this array is stored in \( \text{handle} \) for later use by \( \text{trnlsp_solve} \).

\( \text{iter1} \)

 Specifies the maximum number of iterations.

\( \text{iter2} \)

 Specifies the maximum number of iterations of trial step calculation.

\( rs \)

Definition of initial size of the trust region (boundary of the trial step). The recommend minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set \( rs \) to 0.0, the solver uses the default value, which is 100.0.

**Output Parameters**

\( \text{handle} \)

Type \( \_\text{TRNSP\_HANDLE\_t} \).

\( \text{res} \)

Indicates task completion status.
• res = TR_SUCCESS - the routine completed the task normally.
• res = TR_INVALID_OPTION - there was an error in the input parameters.
• res = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

See Also
?trnlsp_solve

?trnlsp_check
Checks the correctness of handle and arrays containing Jacobian matrix, objective function, and stopping criteria.

Syntax

MKL_INT strnlsp_check (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* fjac, const float* fvec, const float* eps, MKL_INT* info);

MKL_INT dtrnlsp_check (_TRNSP_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* fjac, const double* fvec, const double* eps, MKL_INT* info);

Include Files
• mkl.h

Description
The ?trnlsp_check routine checks the arrays passed into the solver as input parameters. If an array contains any INF or NaN values, the routine sets the flag in output array info(see the description of the values returned in the Output Parameters section for the info array).

Input Parameters

handle  Type _TRNSPBC_HANDLE_t.

n          Length of x.

m          Length of F(x).

fjac     Array of size m by n. Contains the Jacobian matrix of the function.

fvec     Array of size m. Contains the function values at x, where fvec[i] = (y_i - f_i(x)).

es     Array of size 6; contains stopping criteria. See the values in the Description section of the ?trnlsp_init.

Output Parameters

info     Array of size 6.

Results of input parameter checking:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used for</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[0]</td>
<td>Flags for handle</td>
<td>0</td>
<td>The handle is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The handle is not allocated.</td>
</tr>
<tr>
<td>info[1]</td>
<td>Flags for fjac</td>
<td>0</td>
<td>The fjac array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The fjac array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The fjac array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The fjac array contains Inf.</td>
</tr>
<tr>
<td>info[2]</td>
<td>Flags for fvec</td>
<td>0</td>
<td>The fvec array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The fvec array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The fvec array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The fvec array contains Inf.</td>
</tr>
<tr>
<td>info[3]</td>
<td>Flags for eps</td>
<td>0</td>
<td>The eps array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The eps array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The eps array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The eps array contains Inf.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>The eps array contains a value less than or equal to zero.</td>
</tr>
</tbody>
</table>

Information about completion of the task.

res = TR_SUCCESS - the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.h include file.

?trnlsp_solve

Solves a nonlinear least squares problem using the TR algorithm.

Syntax

MKL_INT strnlsp_solve (_TRNSP_HANDLE_t* handle, float* fvec, float* fjac, MKL_INT* RCI_Request);

MKL_INT dtrnlsp_solve (_TRNSP_HANDLE_t* handle, double* fvec, double* fjac, MKL_INT* RCI_Request);

Include Files

- mkl.h

Description

The ?trnlsp_solve routine uses the TR algorithm to solve nonlinear least squares problems.
The problem is stated as follows:

$$\min_{x \in \mathbb{R}^n} \left\| F(x) \right\|^2_2 = \min_{x \in \mathbb{R}^n} \left\| f - f(x) \right\|^2_2, \quad x \in \mathbb{R}^n, \quad f: \mathbb{R}^n \rightarrow \mathbb{R}^m, \quad m \geq n,$$

where

- \( F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m \)
- \( m \geq n \)

From a current point \( x_{\text{current}} \), the algorithm uses the trust-region approach:

$$\min_{x \in \mathbb{R}^n} \left\| F(x_{\text{current}}) + J(x_{\text{current}})(x_{\text{new}} - x_{\text{current}}) \right\|^2_2 \quad \text{subject to} \quad \left\| x_{\text{new}} - x_{\text{current}} \right\|_2 \leq \Delta_{\text{current}}$$

to get \( x_{\text{new}} = x_{\text{current}} + s \) that satisfies

$$\min_{x \in \mathbb{R}^n} \left\| J^T(x)J(x)s + J^T F(x) \right\|^2_2$$

where

- \( J(x) \) is the Jacobian matrix
- \( s \) is the trial step
- \( \|s\|_2 \leq \Delta_{\text{current}} \)

The \textit{RCI\_Request} parameter provides additional information:

<table>
<thead>
<tr>
<th>\textit{RCI_Request Value}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Request to calculate the Jacobian matrix and put the result into \textit{fjac}</td>
</tr>
<tr>
<td>1</td>
<td>Request to recalculate the function at vector ( x ) and put the result into \textit{fvec}</td>
</tr>
<tr>
<td>0</td>
<td>One successful iteration step on the current trust-region radius (that does not mean that the value of ( x ) has changed)</td>
</tr>
<tr>
<td>-1</td>
<td>The algorithm has exceeded the maximum number of iterations</td>
</tr>
<tr>
<td>-2</td>
<td>( \Delta &lt; \text{eps}[0] )</td>
</tr>
<tr>
<td>-3</td>
<td>( | F(x) |_2 &lt; \text{eps}[1] )</td>
</tr>
<tr>
<td>-4</td>
<td>The Jacobian matrix is singular. ( | J(x) |_{m \times (j-1) \ldots m \times j-1} |_2 &lt; \text{eps}[2], \ j = 1, \ldots, n )</td>
</tr>
<tr>
<td>-5</td>
<td>( | s |_2 &lt; \text{eps}[3] )</td>
</tr>
<tr>
<td>-6</td>
<td>( | F(x) |_2 - | F(x) - J(x)s |_2 &lt; \text{eps}[4] )</td>
</tr>
</tbody>
</table>

Note:

- \( J(x) \) is the Jacobian matrix.
- \( \Delta \) is the trust-region area.
- \( F(x) \) is the value of the functional.
- \( s \) is the trial step.

**Input Parameters**

\( \text{handle} \quad \text{Type} \quad \_\text{TRNSP}\_\text{HANDLE}\_\text{t}. \)
**fvec**

Array of size $m$. Contains the function values at $x$, where $fvec[i] = (y_i - f_i(x))$.

**fjac**

Array of size $m$ by $n$. Contains the Jacobian matrix of the function.

**Output Parameters**

**fvec**

Array of size $m$. Updated function evaluated at $x$.

**RCI_Request**

Informs about the task stage.

See the Description section for the parameter values and their meaning.

**res**

Indicates the task completion.

$res = \text{TR\_SUCCESS} - \text{the routine completed the task normally.}$

$\text{TR\_SUCCESS}$ is defined in the mkl_rci.h include file.

---

**trnlsp_get**

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

**Syntax**

```c
MKL_INT strnlsp_get(_TRNSP_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, float* r1, float* r2);
MKL_INT dtrnlsp_get(_TRNSP_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, double* r1, double* r2);
```

**Include Files**

- mkl.h

**Description**

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The initial residual is the value of the functional $\| \| y - f(x) \| \|_2$ of the initial $x$ values provided by the user.

The final residual is the value of the functional $\| \| y - f(x) \| \|_2$ of the final $x$ resulting from the algorithm operation.

The $st\_cr$ parameter contains a number indicating the stop criterion:

<table>
<thead>
<tr>
<th>$st_cr$ Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The algorithm has exceeded the maximum number of iterations</td>
</tr>
<tr>
<td>2</td>
<td>$\Delta &lt; \text{eps}[0]$</td>
</tr>
<tr>
<td>3</td>
<td>$| F(x) |_2 &lt; \text{eps}[1]$</td>
</tr>
<tr>
<td>4</td>
<td>The Jacobian matrix is singular. $| J(x)_{r \times (j-1)\ldots m \times (j-1)} |_2 &lt; \text{eps}[2]$, $j = 1, \ldots, n$</td>
</tr>
<tr>
<td>5</td>
<td>$| s |_2 &lt; \text{eps}[3]$</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>6</td>
<td>$|F(x)|_2 - |F(x) - J(x)s|_2 &lt; \varepsilon[4]$</td>
</tr>
</tbody>
</table>

Note:
- $J(x)$ is the Jacobian matrix.
- $\Delta$ is the trust-region area.
- $F(x)$ is the value of the functional.
- $s$ is the trial step.

### Input Parameters
- **handle**
  - Type: _TRNSP_HANDLE_t.

### Output Parameters
- **iter**
  - Contains the current number of iterations.
- **st_cr**
  - Contains the stop criterion.
  - See the Description section for the parameter values and their meanings.
- **r1**
  - Contains the residual, $(\|y - f(x)\|)$ given the initial $x$.
- **r2**
  - Contains the final residual, that is, the value of the functional $(\|y - f(x)\|)$ of the final $x$ resulting from the algorithm operation.
- **res**
  - Indicates the task completion.
  - $res = TR\_SUCCESS$ - the routine completed the task normally.
  - $TR\_SUCCESS$ is defined in the mkl.rci.h include file.

### strnlsp_delete
*Releases allocated data.*

**Syntax**

```c
MKL\_INT strnlsp\_delete(_TRNSP\_HANDLE\_t* handle);
MKL\_INT dstrnlsp\_delete(_TRNSP\_HANDLE\_t* handle);
```

**Include Files**

- mkl.h

**Description**

The `strnlsp_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by $x$ and $\varepsilon$.

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_freeBuffers`.

### Input Parameters
- **handle**
  - Type: _TRNSP_HANDLE_t.
Output Parameters

Indicates the task completion.

\( res = TR\_SUCCESS \) means the routine completed the task normally.

TR_SUCCESS is defined in the mkl_rci.h include file.

Nonlinear Least Squares Problem with Linear (Bound) Constraints

The nonlinear least squares problem with linear bound constraints is very similar to the nonlinear least squares problem without constraints but it has the following constraints:

\[
I_i \leq x_i \leq U_i, i = 1, \ldots, n, \quad I, U \in \mathbb{R}^n.
\]

See usage examples in the examples/solverc/source folder of your Intel® MKL directory. Specifically, see ex_nlsqp_bc.c.c.

RCI TR Routines for Problem with Bound Constraints

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>?trnlspbc_init</td>
<td>Initializes the solver.</td>
</tr>
<tr>
<td>?trnlspbc_check</td>
<td>Checks correctness of the input parameters.</td>
</tr>
<tr>
<td>?trnlspbc_solve</td>
<td>Solves a nonlinear least squares problem using RCI and the Trust-Region algorithm.</td>
</tr>
<tr>
<td>?trnlspbc_get</td>
<td>Retrieves the number of iterations, stop criterion, initial residual, and final residual.</td>
</tr>
<tr>
<td>?trnlspbc_delete</td>
<td>Releases allocated data.</td>
</tr>
</tbody>
</table>

?trnlspbc_init

Initializes the solver of nonlinear least squares problem with linear (boundary) constraints.

Syntax

MKL_INT strnlspbc_init (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* x, const float* LW, const float* UP, const float* eps, const MKL_INT* iter1, const MKL_INT* iter2, const float* rs);

MKL_INT dtrnlspbc_init (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* x, const double* LW, const double* UP, const double* eps, const MKL_INT* iter1, const MKL_INT* iter2, const double* rs);

Description

The ?trnlspbc_init routine initializes the solver.

After initialization, all subsequent invocations of the ?trnlspbc_solve routine should use the values of the handle returned by ?trnlspbc_init. This handle stores internal data, including pointers to the arrays \( x, LW, UP, \) and \( eps \). It is important to not move or deallocate these arrays until after calling the ?trnlspbc_delete routine.
The `eps` array contains a number indicating the stopping criteria:

<table>
<thead>
<tr>
<th><code>eps</code> Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\Delta &lt; \text{eps}[0]$</td>
</tr>
<tr>
<td>1</td>
<td>$</td>
</tr>
<tr>
<td>2</td>
<td>The Jacobian matrix is singular. $</td>
</tr>
<tr>
<td>3</td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
</tr>
<tr>
<td>5</td>
<td>The trial step precision. If $\text{eps}[5] = 0$, then the trial step meets the required precision ($\leq 1.0 \times 10^{-10}$).</td>
</tr>
</tbody>
</table>

Note:
- $J(x)$ is the Jacobian matrix.
- $\Delta$ is the trust-region area.
- $F(x)$ is the value of the functional.
- $s$ is the trial step.

### Input Parameters

- $n$ Length of $x$.
- $m$ Length of $F(x)$.
- $x$ Array of size $n$. Initial guess. A reference to this array is stored in handle for later use and modification by `?trnlspbc_solve`.
- $LW$ Array of size $n$. Contains low bounds for $x$ ($lw_i < x_i$). A reference to this array is stored in handle for later use by `?trnlspbc_solve`.
- $UP$ Array of size $n$. Contains upper bounds for $x$ ($up_i > x_i$). A reference to this array is stored in handle for later use by `?trnlspbc_solve`.
- $\text{eps}$ Array of size 6; contains stopping criteria. See the values in the Description section. A reference to this array is stored in handle for later use by `?trnlspbc_solve`.
- $\text{iter1}$ Specifies the maximum number of iterations.
- $\text{iter2}$ Specifies the maximum number of iterations of trial step calculation.
- $rs$ Definition of initial size of the trust region (boundary of the trial step). The recommended minimum value is 0.1, and the recommended maximum value is 100.0. Based on your knowledge of the objective function and initial guess you can increase or decrease the initial trust region. It can influence the iteration process, for example, the direction of the iteration process and the number of iterations. If you set $rs$ to 0.0, the solver uses the default value, which is 100.0.
Output Parameters

**handle**
Type _TRNSPBC_HANDLE_t.

**res**
Informs about the task completion.

- res = TR_SUCCESS - the routine completed the task normally.
- res = TR_INVALID_OPTION - there was an error in the input parameters.
- res = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

?trnlspbc_check
Checks the correctness of handle and arrays containing Jacobian matrix, objective function, lower and upper bounds, and stopping criteria.

**Syntax**

MKL_INT strnlspbc_check (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* fjac, const float* fvec, const float* LW, const float* UP, const float* eps, MKL_INT* info);

MKL_INT dtrnlspbc_check (_TRNSPBC_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* fjac, const double* fvec, const double* LW, const double* UP, const double* eps, MKL_INT* info);

**Include Files**

- mkl.h

**Description**
The ?trnlspbc_check routine checks the arrays passed into the solver as input parameters. If an array contains any INF or NaN values, the routine sets the flag in output array info (see the description of the values returned in the Output Parameters section for the info array).

**Input Parameters**

**handle**
Type _TRNSPBC_HANDLE_t.

**n**
Length of x.

**m**
Length of F(x).

**fjac**
Array of size m by n. Contains the Jacobian matrix of the function.

**fvec**
Array of size m. Contains the function values at x, where fvec[i] = (y_i - f_i(x)).

**LW**
Array of size n.
Contains low bounds for x (lw_i < x_i).

**UP**
Array of size n.
Contains upper bounds for x (up_i > x_i).
**eps**

Array of size 6; contains stopping criteria. See the values in the Description section of the ?trnlspbc_init.

**Output Parameters**

**info**

Array of size 6.

Results of input parameter checking:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Used for</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>info[0]</td>
<td>Flags for ( handle )</td>
<td>0</td>
<td>The handle is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The handle is not allocated.</td>
</tr>
<tr>
<td>info[1]</td>
<td>Flags for ( fjac )</td>
<td>0</td>
<td>The ( fjac ) array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The ( fjac ) array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The ( fjac ) array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The ( fjac ) array contains Inf.</td>
</tr>
<tr>
<td>info[2]</td>
<td>Flags for ( fvec )</td>
<td>0</td>
<td>The ( fvec ) array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The ( fvec ) array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The ( fvec ) array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The ( fvec ) array contains Inf.</td>
</tr>
<tr>
<td>info[3]</td>
<td>Flags for ( LW )</td>
<td>0</td>
<td>The ( LW ) array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The ( LW ) array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The ( LW ) array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The ( LW ) array contains Inf.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>The lower bound is greater than the upper bound.</td>
</tr>
<tr>
<td>info[4]</td>
<td>Flags for ( up )</td>
<td>0</td>
<td>The ( up ) array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The ( up ) array is not allocated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>The ( up ) array contains NaN.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>The ( up ) array contains Inf.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>The upper bound is less than the lower bound.</td>
</tr>
<tr>
<td>info[5]</td>
<td>Flags for ( eps )</td>
<td>0</td>
<td>The ( eps ) array is valid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>The ( eps ) array is not allocated</td>
</tr>
</tbody>
</table>
Parameter | Used for | Value | Description
--- | --- | --- | ---
| | | 2 | The \( \text{eps} \) array contains NaN.
| | | 3 | The \( \text{eps} \) array contains Inf.
| | | 4 | The \( \text{eps} \) array contains a value less than or equal to zero.

\( \text{res} \) Information about completion of the task.

\( \text{res} = \text{TR\_SUCCESS} \) - the routine completed the task normally.

\text{TR\_SUCCESS} \ is defined in \text{mkl\_rci.h} include file.

\(?\text{trnlspbc\_solve}\)

Solves a nonlinear least squares problem with linear (bound) constraints using the Trust-Region algorithm.

Syntax

\[
\text{MKL\_INT \?\text{trnlspbc\_solve} (\_\text{TRNSPBC\_HANDLE\_t * handle, float* fvec, float* fjac, MKL\_INT* RCI\_Request});}
\]

\[
\text{MKL\_INT d\?trnlspbc\_solve (\_\text{TRNSPBC\_HANDLE\_t * handle, double* fvec, double* fjac, MKL\_INT* RCI\_Request});}
\]

Include Files

• \text{mkl.h}

Description

The \(?\text{trnlspbc\_solve}\) routine, based on RCI, uses the Trust-Region algorithm to solve nonlinear least squares problems with linear (bound) constraints. The problem is stated as follows:

\[
\min_{x \in \mathbb{R}^n} \| R(x) \|_2^2 = \min_{x \in \mathbb{R}^n} \| y - f(x) \|_2^2, \quad y \in \mathbb{R}^m, \quad x \in \mathbb{R}^n, \quad f : \mathbb{R}^n \to \mathbb{R}^m, \quad m \geq n
\]

where

\( l_i \leq x_i \leq u_i \)

\( i = 1, \ldots, n. \)

The \text{RCI\_Request} parameter provides additional information:

<table>
<thead>
<tr>
<th>\text{RCI_Request} Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Request to calculate the Jacobian matrix and put the result into \text{fjac}</td>
</tr>
<tr>
<td>1</td>
<td>Request to recalculate the function at vector ( x ) and put the result into \text{fvec}</td>
</tr>
<tr>
<td>0</td>
<td>One successful iteration step on the current trust-region radius (that does not mean that the value of ( x ) has changed)</td>
</tr>
<tr>
<td>-1</td>
<td>The algorithm has exceeded the maximum number of iterations</td>
</tr>
</tbody>
</table>
RCI_Request | Value | Description
--- | --- | ---
-2 | $\Delta < \text{eps}[0]$ | 
-3 | $||F(x)||_2 < \text{eps}[1]$ | 
-4 | The Jacobian matrix is singular. $||J(x)[m^*(j-1)\ldots m^*j-1]||_2 < \text{eps}[2], j = 1, \ldots, n$ | 
-5 | $||s||_2 < \text{eps}[3]$ | 
-6 | $||F(x)||_2 - ||F(x) - J(x)s||_2 < \text{eps}[4]$ | 

Note:
- $J(x)$ is the Jacobian matrix.
- $\Delta$ is the trust-region area.
- $F(x)$ is the value of the functional.
- $s$ is the trial step.

**Input Parameters**

*handle* Type _TRNSPBC_HANDLE_t.

*fvec* Array of size $m$. Contains the function values at $x$, where $fvec[i] = (y_i - f_i(x))$.

*fjac* Array of size $m$ by $n$. Contains the Jacobian matrix of the function.

**Output Parameters**

*fvec* Array of size $m$. Updated function evaluated at $x$.

RCI_Request Informs about the task stage.

See the Description section for the parameter values and their meaning.

*res* Informs about the task completion.

$res = \text{TR\_SUCCESS}$ means the routine completed the task normally. 

**trnlspbc_get**

Retrieves the number of iterations, stop criterion, initial residual, and final residual.

**Syntax**

MKL_INT strnlspbc_get (_TRNSPBC_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, float* r1, float* r2);

MKL_INT dtrnlspbc_get (_TRNSPBC_HANDLE_t* handle, MKL_INT* iter, MKL_INT* st_cr, double* r1, double* r2);

**Include Files**
- mkl.h
Description

The routine retrieves the current number of iterations, the stop criterion, the initial residual, and final residual.

The \textit{st\_cr} parameter contains a number indicating the stop criterion:

<table>
<thead>
<tr>
<th>\textit{st_cr} Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The algorithm has exceeded the maximum number of iterations</td>
</tr>
<tr>
<td>2 ( \Delta &lt; \text{eps}[0] )</td>
<td></td>
</tr>
<tr>
<td>3 ( |F(x)|_2 &lt; \text{eps}[1] )</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>The Jacobian matrix is singular.</td>
</tr>
<tr>
<td></td>
<td>( |J(x)_{[m^{<em>}(j-1)\ldots m^{</em>}j-1]}|_2 &lt; \text{eps}[2], \ j = 1, \ldots, n )</td>
</tr>
<tr>
<td>5 ( |s|_2 &lt; \text{eps}[3] )</td>
<td></td>
</tr>
<tr>
<td>6 ( |F(x)|_2 - |F(x) - J(x)s|_2 &lt; \text{eps}[4] )</td>
<td></td>
</tr>
</tbody>
</table>

Note:
- \( J(x) \) is the Jacobian matrix.
- \( \Delta \) is the trust-region area.
- \( F(x) \) is the value of the functional.
- \( s \) is the trial step.

Input Parameters

- \textit{handle}  
  Type \_TRNSPBC\_HANDLE\_t.

Output Parameters

- \textit{iter}  
  Contains the current number of iterations.
- \textit{st\_cr}  
  Contains the stop criterion. See the Description section for the parameter values and their meanings.
- \textit{r1}  
  Contains the residual, \( \|y - f(x)\| \) given the initial \( x \).
- \textit{r2}  
  Contains the final residual, that is, the value of the function \( \|y - f(x)\| \) of the final \( x \) resulting from the algorithm operation.
- \textit{res}  
  Informs about the task completion.

\[ \text{res} = \text{TR\_SUCCESS} - \text{the routine completed the task normally.} \]

\[ \text{TR\_SUCCESS is defined in the mkl\_rci.h include file.} \]

\?trnlspbc_delete

Releases allocated data.

Syntax

\begin{verbatim}
MKL\_INT strnlspbc\_delete (\_TRNSPBC\_HANDLE\_t* handle);
MKL\_INT dstrnlspbc\_delete (\_TRNSPBC\_HANDLE\_t* handle);
\end{verbatim}
Include Files

- mkl.h

Description

The `?trnlspbc_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by `x`, `LW`, `UP`, and `eps`.

**NOTE**

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_free_buffers`.

Input Parameters

- **handle**
  - Type: `_TRNSPBC_HANDLE_t`.

Output Parameters

- **res**
  - Informs about the task completion.
  - `res = TR_SUCCESS` means the routine completed the task normally.
  - `TR_SUCCESS` is defined in the `mkl_rci.h` include file.

Jacobian Matrix Calculation Routines

This section describes routines that compute the Jacobian matrix using the central difference algorithm. Jacobian matrix calculation is required to solve a nonlinear least squares problem and systems of nonlinear equations (with or without linear bound constraints). Routines for calculation of the Jacobian matrix have the "Black-Box" interfaces, where you pass the objective function via parameters. Your objective function must have a fixed interface.

### Jacobian Matrix Calculation Routines

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?jacobi_init</code></td>
<td>Initializes the solver.</td>
</tr>
<tr>
<td><code>?jacobi_solve</code></td>
<td>Computes the Jacobian matrix of the function on the basis of RCI using the central difference algorithm.</td>
</tr>
<tr>
<td><code>?jacobi_delete</code></td>
<td>Removes data.</td>
</tr>
<tr>
<td><code>?jacobi</code></td>
<td>Computes the Jacobian matrix of the <code>fcn</code> function using the central difference algorithm.</td>
</tr>
<tr>
<td><code>?jacobix</code></td>
<td>Presents an alternative interface for the <code>?jacobi</code> function enabling you to pass additional data into the objective function.</td>
</tr>
</tbody>
</table>

### `?jacobi_init`

*Initializes the solver for Jacobian calculations.*
Syntax

MKL_INT sjacobi_init (_JACOBIMATRIX_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const float* x, const float* fjac, const float* eps);

MKL_INT djacobi_init (_JACOBIMATRIX_HANDLE_t* handle, const MKL_INT* n, const MKL_INT* m, const double* x, const double* fjac, const double* eps);

Include Files

• mkl.h

Description

The routine initializes the solver.

Input Parameters

- **n**
  - Length of x.

- **m**
  - Length of F.

- **x**
  - Array of size n. Vector, at which the function is evaluated.

  A reference to this array is stored in handle for later use and modification by ?jacobi_solve.

- **eps**
  - Precision of the Jacobian matrix calculation.

- **fjac**
  - Array of size m by n. Contains the Jacobian matrix of the function.

  A reference to this array is stored in handle for later use and modification by ?jacobi_solve.

Output Parameters

- **handle**
  - Data object of the _JACOBIMATRIX_HANDLE_t type. Stores internal data, including pointers to the user-provided arrays x and fjac. It is important that the user does not move or deallocate these arrays until after calling the ?jacobi_delete routine.

- **res**
  - Indicates task completion status.
  - res = TR_SUCCESS - the routine completed the task normally.
  - res = TR_INVALID_OPTION - there was an error in the input parameters.
  - res = TR_OUT_OF_MEMORY - there was a memory error.

  TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

?jacobi_solve

Computes the Jacobian matrix of the function using RCI and the central difference algorithm.

Syntax

MKL_INT sjacobi_solve (_JACOBIMATRIX_HANDLE_t* handle, float* f1, float* f2, MKL_INT* RCI_Request);
MKL_INT djacobi_solve (_JACOBIMATRIX_HANDLE_t* handle, double* f1, double* f2, MKL_INT* RCI_Request);

**Include Files**
- mkl.h

**Description**
The `?jacobi_solve` routine computes the Jacobian matrix of the function using RCI and the central difference algorithm.

See usage examples in the `examples\solverc\source` folder of your Intel® MKL directory. Specifically, see `sjacobi_rci_c.c` and `djacobi_rci_c.c`.

**Input Parameters**
- **handle**: Type _JACOBIMATRIX_HANDLE_t.
- **RCI_Request**: Set to 0 before the first call to `?jacobi_solve`.

**Output Parameters**
- **f1**: Contains the updated function values at `x + eps`.
- **f2**: Array of size `m`. Contains the updated function values at `x - eps`.
- **RCI_Request**: Provides information about the task completion. When equal to 0, the task has completed successfully.
  - `RCI_Request` = 1 indicates that you should compute the function values at the current `x` point and put the results into `f1`.
  - `RCI_Request` = 2 indicates that you should compute the function values at the current `x` point and put the results into `f2`.
- **res**: Indicates the task completion status.
  - `res` = TR_SUCCESS - the routine completed the task normally.
  - `res` = TR_INVALID_OPTION - there was an error in the input parameters.

TR_SUCCESS and TR_INVALID_OPTION are defined in the mkl_rci.h include file.

**See Also**
- `?jacobi_init`
- `?jacobi_delete`

Releases allocated data.

**Syntax**
- MKL_INT sjacobi_delete (_JACOBIMATRIX_HANDLE_t* handle);
- MKL_INT djacobi_delete (_JACOBIMATRIX_HANDLE_t* handle);
Include Files

- mkl.h

Description

The `?jacobi_delete` routine releases all memory allocated for the handle. Only after calling this routine is it safe for the user to move or deallocate the memory referenced by `x` and `fjac`.

This routine flags memory as not used, but to actually release all memory you must call the support function `mkl_freeBuffers`.

Input Parameters

`handle`  
Type: `_JACOBIMATRIX_HANDLE_t`.

Output Parameters

`res`  
Informs about the task completion.

`res = TR_SUCCESS` means the routine completed the task normally.

`TR_SUCCESS` is defined in the `mkl_rci.h` include file.

`?jacobi`

*Computes the Jacobian matrix of the objective function using the central difference algorithm.*

Syntax

```c
MKL_INT sjacobi (USRFCNS fcn, const MKL_INT* n, const MKL_INT* m, float* fjac, float* x, float* eps);
MKL_INT djacobi (USRFCND fcn, const MKL_INT* n, const MKL_INT* m, double* fjac, double* x, double* eps);
```

Include Files

- mkl.h

Description

The `?jacobi` routine computes the Jacobian matrix for function `fcn` using the central difference algorithm. This routine has a "Black-Box" interface, where you input the objective function via parameters. Your objective function must have a fixed interface.

See calling and usage examples in the examples\solverc\source folder of your Intel® MKL directory. Specifically, see `ex_nlsqp_c.c` and `ex_nlsqp_bc_c.c`.

Input Parameters

`fcn`  
User-supplied subroutine to evaluate the function that defines the least squares problem. Called as `fcn (m, n, x, f)` with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>m</code></td>
<td>Pointer to the length of <code>f</code>.</td>
<td></td>
</tr>
</tbody>
</table>
Parameter | Type | Description
--- | --- | ---
\( n \) | Pointer to the length of \( x \).
\( x \) | Array of size \( n \). Vector, at which the function is evaluated. The \( fcn \) function should not change this parameter.

Output Parameters
\( f \) | Array of size \( m \); contains the function values at \( x \).

You need to declare \( fcn \) as \texttt{extern} in the calling program.

\( n \) | Length of \( X \).
\( m \) | Length of \( F \).
\( x \) | Array of size \( n \). Vector at which the function is evaluated.
\( \text{eps} \) | Precision of the Jacobian matrix calculation.

Output Parameters
\( fjac \) | Array of size \( m \) by \( n \). Contains the Jacobian matrix of the function.
\( \text{res} \) | Indicates task completion status.
  * \( \text{res} = \text{TR_SUCCESS} \) - the routine completed the task normally.
  * \( \text{res} = \text{TR_INVALID_OPTION} \) - there was an error in the input parameters.
  * \( \text{res} = \text{TR_OUT_OF_MEMORY} \) - there was a memory error.

\text{TR_SUCCESS}, \text{TR_INVALID_OPTION}, and \text{TR_OUT_OF_MEMORY} are defined in the \texttt{mkl_rci.h} include file.

See Also
\?jacobix

\?jacobi

Alternative interface for \?jacobi function for passing additional data into the objective function.

Syntax

\begin{verbatim}
MKL_INT sjacobix (USRFCNXS fcn, const MKL_INT* n, const MKL_INT* m, float* fjac, float* x, float* eps, void* user_data);
MKL_INT djacobix (USRFCNXD fcn, const MKL_INT* n, const MKL_INT* m, double* fjac, double* x, double* eps, void* user_data);
\end{verbatim}

Include Files

* \texttt{mkl.h}

Description

The \?jacobi routine presents an alternative interface for the \?jacobi function that enables you to pass additional data into the objective function \( fcn \).
See calling and usage examples in the examples\solverc\source folder of your Intel® MKL directory. Specifically, see ex_nlsqp_c_x.c and ex_nlsqp_bc_c_x.c.

**Input Parameters**

*fcn*  
User-supplied subroutine to evaluate the function that defines the least squares problem. Called as *fcn* \((m, n, x, f, \text{user\_data})\) with the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Parameters</strong></td>
<td></td>
</tr>
<tr>
<td><em>m</em></td>
<td>Pointer to the length of (f).</td>
</tr>
<tr>
<td><em>n</em></td>
<td>Pointer to the length of (x).</td>
</tr>
<tr>
<td><em>x</em></td>
<td>Array of size (n). Vector at which the function is evaluated. The <em>fcn</em> function should not change this parameter.</td>
</tr>
<tr>
<td><em>user_data</em></td>
<td>Pointer to your additional data, if any. Otherwise, a dummy argument.</td>
</tr>
<tr>
<td><strong>Output Parameters</strong></td>
<td></td>
</tr>
<tr>
<td><em>f</em></td>
<td>Array of size (m); contains the function values at (x).</td>
</tr>
</tbody>
</table>

You need to declare *fcn* as *extern* in the calling program.

*n*  
Length of \(X\).

*m*  
Length of \(F\).

*x*  
Array of size \(n\). Vector at which the function is evaluated.

*eps*  
Precision of the Jacobian matrix calculation.

*user\_data*  
Pointer to your additional data. If there is no additional data, this is a dummy argument.

**Output Parameters**

*fjac*  
Array of size \(m\) by \(n\). Contains the Jacobian matrix of the function.

*res*  
Indicates task completion status.

- *res* = TR_SUCCESS - the routine completed the task normally.
- *res* = TR_INVALID_OPTION - there was an error in the input parameters.
- *res* = TR_OUT_OF_MEMORY - there was a memory error.

TR_SUCCESS, TR_INVALID_OPTION, and TR_OUT_OF_MEMORY are defined in the mkl_rci.h include file.

**See Also**

?jacobi
Support Functions

Intel® Math Kernel Library (Intel® MKL) support functions are subdivided into the following groups according to their purpose:

- Version Information
- Threading Control
- Error Handling
- Character Equality Testing
- Timing
- Memory Management
- Single Dynamic Library Control
- Conditional Numerical Reproducibility Control
- Miscellaneous

The following table lists Intel® MKL support functions.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version Information</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_get_version</td>
<td>Returns the Intel® MKL version.</td>
</tr>
<tr>
<td>mkl_get_version_string</td>
<td>Returns the Intel® MKL version in a character string.</td>
</tr>
<tr>
<td><strong>Threading Control</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_set_num_threads</td>
<td>Specifies the number of OpenMP* threads to use.</td>
</tr>
<tr>
<td>mkl_domain_set_num_threads</td>
<td>Specifies the number of OpenMP* threads for a particular function domain.</td>
</tr>
<tr>
<td>mkl_set_num_threads_local</td>
<td>Specifies the number of OpenMP* threads for all Intel® MKL functions on the current execution thread.</td>
</tr>
<tr>
<td>mkl_set_dynamic</td>
<td>Enables Intel® MKL to dynamically change the number of OpenMP* threads.</td>
</tr>
<tr>
<td>mkl_get_max_threads</td>
<td>Gets the number of OpenMP* threads targeted for parallelism.</td>
</tr>
<tr>
<td>mkl_domain_get_max_threads</td>
<td>Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.</td>
</tr>
<tr>
<td>mkl_get_dynamic</td>
<td>Determines whether Intel® MKL is enabled to dynamically change the number of OpenMP* threads.</td>
</tr>
<tr>
<td>mkl_set_num_stripes</td>
<td>Specifies the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.</td>
</tr>
<tr>
<td>mkl_get_num_stripes</td>
<td>Gets the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Operation</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Error Handling</strong></td>
<td></td>
</tr>
<tr>
<td>xerbla</td>
<td>Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.</td>
</tr>
<tr>
<td>pxerbla</td>
<td>Handles error conditions for the ScaLAPACK routines.</td>
</tr>
<tr>
<td>LAPACKE_xerbla</td>
<td>Error handling function called by the C interface to LAPACK functions.</td>
</tr>
<tr>
<td>mkl_set_exit_handler</td>
<td>Sets the custom handler of fatal errors.</td>
</tr>
<tr>
<td><strong>Character Equality Testing</strong></td>
<td></td>
</tr>
<tr>
<td>lsame</td>
<td>Tests two characters for equality regardless of the case.</td>
</tr>
<tr>
<td>lsamen</td>
<td>Tests two character strings for equality regardless of the case.</td>
</tr>
<tr>
<td><strong>Timing</strong></td>
<td></td>
</tr>
<tr>
<td>second/dsecnd</td>
<td>Returns elapsed time in seconds. Use to estimate real time between two calls to this function.</td>
</tr>
<tr>
<td>mkl_get_cpu_clocks</td>
<td>Returns elapsed CPU clocks.</td>
</tr>
<tr>
<td>mkl_get_cpu_frequency</td>
<td>Returns CPU frequency value in GHz.</td>
</tr>
<tr>
<td>mkl_get_max_cpu_frequency</td>
<td>Returns the maximum CPU frequency value in GHz.</td>
</tr>
<tr>
<td>mkl_get_clocks_frequency</td>
<td>Returns the frequency value in GHz based on constant-rate Time Stamp Counter.</td>
</tr>
<tr>
<td><strong>Memory Management</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_freeBuffers</td>
<td>Frees unused memory allocated by the Intel® MKL Memory Allocator.</td>
</tr>
<tr>
<td>mkl_thread_freeBuffers</td>
<td>Frees unused memory allocated by the Intel® MKL Memory Allocator in the current thread.</td>
</tr>
<tr>
<td>mkl_mem_stat</td>
<td>Reports the status of the Intel® MKL Memory Allocator.</td>
</tr>
<tr>
<td>mkl_peak_mem_usage</td>
<td>Reports the peak memory allocated by the Intel® MKL Memory Allocator.</td>
</tr>
<tr>
<td>mkl_disable_fast_mm</td>
<td>Turns off the Intel® MKL Memory Allocator for Intel® MKL functions to directly use the system malloc/free functions.</td>
</tr>
<tr>
<td>mkl_malloc</td>
<td>Allocates an aligned memory buffer.</td>
</tr>
<tr>
<td>mkl_calloc</td>
<td>Allocates and initializes an aligned memory buffer.</td>
</tr>
<tr>
<td>mkl_realloc</td>
<td>Changes the size of memory buffer allocated by mkl_malloc/mkl_calloc.</td>
</tr>
<tr>
<td>mkl_free</td>
<td>Frees the aligned memory buffer allocated by mkl_malloc/mkl_calloc.</td>
</tr>
<tr>
<td>mkl_set_memory_limit</td>
<td>On Linux, sets the limit of memory that Intel® MKL can allocate for a specified type of memory.</td>
</tr>
<tr>
<td>Function Name</td>
<td>Operation</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Single Dynamic Library (SDL) Control</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_set_interface_layer</td>
<td>Sets the interface layer for Intel® MKL at run time.</td>
</tr>
<tr>
<td>mkl_set_threading_layer</td>
<td>Sets the threading layer for Intel® MKL at run time.</td>
</tr>
<tr>
<td>mkl_set_xerbla</td>
<td>Replaces the error handling routine. Use with the Single Dynamic Library.</td>
</tr>
<tr>
<td>mkl_set_progress</td>
<td>Replaces the progress information routine.</td>
</tr>
<tr>
<td>mkl_set_pardiso_pivot</td>
<td>Replaces the routine handling Intel® MKL PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).</td>
</tr>
<tr>
<td><strong>Conditional Numerical Reproducibility (CNR) Control</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_cbwr_set</td>
<td>Configures the CNR mode of Intel® MKL.</td>
</tr>
<tr>
<td>mkl_cbwr_get</td>
<td>Returns the current CNR settings.</td>
</tr>
<tr>
<td>mkl_cbwr_get_auto_branch</td>
<td>Automatically detects the CNR code branch for your platform.</td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
</tr>
<tr>
<td>mkl_progress</td>
<td>Provides progress information.</td>
</tr>
<tr>
<td>mkl_enable_instructions</td>
<td>Enables dispatching for new Intel® architectures or restricts the set of Intel® instruction sets available for dispatching.</td>
</tr>
<tr>
<td>mkl_set_env_mode</td>
<td>Set up the mode that ignores environment settings specific to Intel® MKL.</td>
</tr>
<tr>
<td>mklVerbose</td>
<td>Enable or disable Intel® MKL Verbose mode.</td>
</tr>
<tr>
<td>mklVerbose_output_file</td>
<td>Write output in Intel® MKL Verbose mode to a file.</td>
</tr>
<tr>
<td>mkl_set_mpi</td>
<td>Sets the implementation of the message-passing interface to be used by Intel® MKL.</td>
</tr>
<tr>
<td>mkl_finalize</td>
<td>Terminates Intel® MKL execution environment and frees resources allocated by the library.</td>
</tr>
</tbody>
</table>

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Version Information**

Intel® MKL provides two methods for extracting information about the library version number:
• extracting a version string using the `mkl_get_version_string` function
• using the `mkl_get_version` function to obtain an `MKLVersion` structure that contains the version information

A makefile is also provided to automatically build the examples and output summary files containing the version information for the current library.

**mkl_get_version**

*Returns the Intel® MKL version.*

**Syntax**

```c
void mkl_get_version( MKLVersion* pVersion );
```

**Include Files**

- mkl.h

**Output Parameters**

`pVersion`  
Pointer to the `MKLVersion` structure.

**Description**

The `mkl_get_version` function collects information about the active C version of the Intel® MKL software and returns this information in a structure of `MKLVersion` type by the `pVersion` address. The `MKLVersion` structure type is defined in the `mkl_types.h` file. The following fields of the `MKLVersion` structure are available:

- **MajorVersion**
  is the major number of the current library version.

- **MinorVersion**
  is the minor number of the current library version.

- **UpdateVersion**
  is the update number of the current library version.

- **ProductStatus**
  is the status of the current library version. Possible variants are "Beta" or "Product".

- **Build**
  is the string that contains the build date and the internal build number.

- **Platform**
  is the string that contains the current architecture. Possible variants are "IA-32 architecture" or "Intel(R) 64 architecture".

- **Processor**
  is the processor optimization. Normally it is targeted for the processor installed on your system and based on the detection of the Intel® MKL library that is optimal for the installed processor. In the Conditional Numerical Reproducibility (CNR) mode, the processor optimization matches the selected CNR branch.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
### Optimization Notice

Optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### mkl_get_version Usage

-----------------------------------------------------------------------------------------------

```c
#include <stdio.h>
#include <stdlib.h>
#include "mkl.h"

int main(void)
{
    MKLVersion Version;

    mkl_get_version(&Version);

    printf("Major version:           %d\n",Version.MajorVersion);
    printf("Minor version:           %d\n",Version.MinorVersion);
    printf("Update version:          %d\n",Version.UpdateVersion);
    printf("Product status:          %s\n",Version.ProductStatus);
    printf("Build:                   %s\n",Version.Build);
    printf("Platform:                %s\n",Version.Platform);
    printf("Processor optimization:  %s\n",Version.Processor);
    printf("================================================================\n");
    printf("\n");
    return 0;
}
```

Output:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Major Version</td>
<td>11</td>
</tr>
<tr>
<td>Minor Version</td>
<td>0</td>
</tr>
<tr>
<td>Update Version</td>
<td>2</td>
</tr>
<tr>
<td>Product status</td>
<td>Product</td>
</tr>
<tr>
<td>Build</td>
<td>20121113</td>
</tr>
<tr>
<td>Platform</td>
<td>Intel(R) 64 architecture</td>
</tr>
<tr>
<td>Processor optimization</td>
<td>Intel(R) Core(TM) i7 Processor</td>
</tr>
</tbody>
</table>

### See Also

Conditional Numerical Reproducibility Control

### mkl_get_version_string

*Returns the Intel® MKL version in a character string.*
Syntax

void mkl_get_version_string (char* buf, int len);

Include Files

• mkl.h

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>char*</td>
<td>Source string</td>
</tr>
<tr>
<td>len</td>
<td>int</td>
<td>Length of the source string</td>
</tr>
</tbody>
</table>

Description

The function returns a string that contains the Intel® MKL version.

For usage details, see the code example below:

Example

```c
#include <stdio.h>
#include "mkl.h"

int main(void)
{
    int len=198;
    char buf[198];
    mkl_get_version_string(buf, len);
    printf("%s
",buf);
    printf("\n");
    return 0;
}
```

Threading Control

Intel® MKL provides functions for OpenMP* threading control, discussed in this section.

**Important**

If Intel® MKL operates within the Intel® Threading Building Blocks (Intel® TBB) execution environment, the environment variables for OpenMP* threading control, such as OMP_NUM_THREADS, and Intel® MKL functions discussed in this section have no effect. If the Intel TBB threading technology is used, control the number of threads through the Intel TBB application programming interface. Read the documentation for the tbb::task_scheduler_init class at https://www.threadingbuildingblocks.org/documentation to find out how to specify the number of Intel TBB threads.

If Intel® MKL operates within an OpenMP* execution environment, you can control the number of threads for Intel® MKL using OpenMP* run-time library routines and environment variables (see the OpenMP* specification for details). Additionally Intel® MKL provides optional threading control functions and environment variables that enable you to specify the number of threads for Intel® MKL and to control dynamic adjustment of the number of threads independently of the OpenMP* settings. The settings made with the Intel® MKL threading control functions and environment variables do not affect OpenMP* settings but take precedence over them.

If none of the threading control functions is used, Intel® MKL environment variables may control Intel® MKL threading. For details of those environment variables, see the Intel® MKL Developer Guide.
You can specify the number of threads for Intel® MKL function domains with the `mkl_set_num_threads` or `mkl_domain_set_num_threads` function. While `mkl_set_num_threads` specifies the number of threads for the entire Intel® MKL, `mkl_domain_set_num_threads` does it for a specific function domain. The following table lists the function domains that support independent threading control. The table also provides named constants to pass to threading control functions as a parameter that specifies the function domain.

**Intel® MKL Function Domains**

<table>
<thead>
<tr>
<th>Function Domain</th>
<th>Named Constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Linear Algebra Subroutines (BLAS)</td>
<td>MKL_DOMAIN_BLAS</td>
</tr>
<tr>
<td>Fast Fourier Transform (FFT) functions, except Cluster FFT functions</td>
<td>MKL_DOMAIN_FFT</td>
</tr>
<tr>
<td>Vector Math (VM) functions</td>
<td>MKL_DOMAIN_VML</td>
</tr>
<tr>
<td>Parallel Direct Solver (PARDISO) functions</td>
<td>MKL_DOMAIN_PARDISO</td>
</tr>
<tr>
<td>All Intel® MKL functions except the functions from the domains where the number of threads is set explicitly</td>
<td>MKL_DOMAIN_ALL</td>
</tr>
</tbody>
</table>

**Warning**

Do not increase the number of OpenMP threads used for cluster_sparse_solver between the first call and the factorization or solution phase. Because the minimum amount of memory required for out-of-core execution depends on the number of OpenMP threads, increasing it after the initial call can cause incorrect results.

Both `mkl_set_num_threads` and `mkl_domain_set_num_threads` functions set the number of threads for all subsequent calls to Intel® MKL from all applications threads. Use the `mkl_set_num_threads_local` function to specify different numbers of threads for Intel® MKL on different execution threads of your application. The thread-local settings take precedence over the global settings. However, the thread-local settings may have undesirable side effects (see the description of the `mkl_set_num_threads_local` function for details).

By default, Intel® MKL can adjust the specified number of threads dynamically. For example, Intel® MKL may use fewer threads if the size of the computation is not big enough or not create parallel regions when running within an OpenMP* parallel region. Although Intel® MKL may actually use a different number of threads from the number specified, the library does not create parallel regions with more threads than specified. If dynamic adjustment of the number of threads is disabled, Intel® MKL attempts to use the specified number of threads in internal parallel regions (for more information, see the Intel® MKL Developer Guide). Use the `mkl_set_dynamic` function to control dynamic adjustment of the number of threads.

**mkl_set_num_threads**

Specifies the number of OpenMP* threads to use.

**Syntax**

```c
void mkl_set_num_threads( int nt );
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| `nt` | int  | `nt > 0` - The number of threads suggested by the user.  
|      |      | `nt \leq 0` - Invalid value, which is ignored. |
Description

This function enables you to specify how many OpenMP threads Intel® MKL should use for internal parallel regions. If this number is not set (default), Intel® MKL functions use the default number of threads for the OpenMP run-time library. The specified number of threads applies:

- To all Intel® MKL functions except the functions from the domains where the number of threads is set with `mkl_domain_set_num_threads`
- To all execution threads except the threads where the number of threads is set with `mkl_set_num_threads_local`

The number specified is a hint, and Intel® MKL may actually use a smaller number.

**NOTE**
This function takes precedence over the `MKL_NUM_THREADS` environment variable.

Example

```c
#include "mkl.h"
...
mkl_set_num_threads(4);
my_compute_using_mkl();    // Intel MKL uses up to 4 OpenMP threads
```

**mkl_domain_set_num_threads**

 Specifies the number of OpenMP* threads for a particular function domain.

Syntax

```c
int mkl_domain_set_num_threads (int nt, int domain);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt</td>
<td>int</td>
<td><code>nt &gt; 0</code> - The number of threads suggested by the user.&lt;br&gt;<code>nt = 0</code> - The default number of threads for the OpenMP run-time library.&lt;br&gt;<code>nt &lt; 0</code> - Invalid value, which is ignored.</td>
</tr>
<tr>
<td>domain</td>
<td>int</td>
<td>The named constant that defines the targeted domain.</td>
</tr>
</tbody>
</table>

Description

This function specifies how many OpenMP threads a particular function domain of Intel® MKL should use. If this number is not set (default) or if it is set to zero in a call to this function, Intel® MKL uses the default number of threads for the OpenMP run-time library. The number of threads specified applies to the specified function domain on all execution threads except the threads where the number of threads is set with `mkl_set_num_threads_local`. For a list of supported values of the `domain` argument, see Table "Intel MKL Function Domains".

The number of threads specified is only a hint, and Intel® MKL may actually use a smaller number.
NOTE
This function takes precedence over the MKL_DOMAIN_NUM_THREADS environment variable.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ierr</td>
<td>int</td>
<td>1 - Indicates no error, execution is successful. 0 - Indicates a failure, possibly because of invalid input parameters.</td>
</tr>
</tbody>
</table>

Example

```c
#include "mkl.h"
...
mkl_domain_set_num_threads(4, MKL_DOMAIN_BLAS);
my_compute_using_mkl_blas();  // Intel MKL BLAS functions use up to 4 threads
my_compute_using_mkl_dft();   // Intel MKL FFT functions use the default number of threads
```

mkl_set_num_threads_local

Specifies the number of OpenMP* threads for all Intel® MKL functions on the current execution thread.

Syntax

```c
int mkl_set_num_threads_local( int nt );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt</td>
<td>int</td>
<td>nt &gt; 0 - The number of threads for Intel® MKL functions to use on the current execution thread. nt = 0 - A request to reset the thread-local number of threads and use the global number.</td>
</tr>
</tbody>
</table>

Description

This function sets the number of OpenMP threads that Intel® MKL functions should request for parallel computation. The number of threads is thread-local, which means that it only affects the current execution thread of the application. If the thread-local number is not set or if this number is set to zero in a call to this function, Intel® MKL functions use the global number of threads. You can set the global number of threads using the mkl_set_num_threads or mkl_domain_set_num_threads function.

The thread-local number of threads takes precedence over the global number: if the thread-local number is non-zero, changes to the global number of threads have no effect on the current thread.
**Caution**
If your application is threaded with OpenMP* and parallelization of Intel® MKL is based on nested OpenMP parallelism, different OpenMP parallel regions reuse OpenMP threads. Therefore a thread-local setting in one OpenMP parallel region may continue to affect not only the master thread after the parallel region ends, but also subsequent parallel regions. To avoid performance implications of this side effect, reset the thread-local number of threads before leaving the OpenMP parallel region (see Examples for how to do it).

**Return Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>save_nt</td>
<td>int</td>
<td>The value of the thread-local number of threads that was used before this function call. Zero means that the global number of threads was used.</td>
</tr>
</tbody>
</table>

**Examples**

This example shows how to avoid the side effect of a thread-local number of threads by reverting to the global setting:

```c
#include "omp.h"
#include "mkl.h"
...
mkl_set_num_threads(16);
my_compute_using_mkl(); // Intel MKL functions use up to 16 threads
#pragma omp parallel num_threads(2)
{
    if (0 == omp_get_thread_num())
        mkl_set_num_threads_local(4);
    else
        mkl_set_num_threads_local(12);

    my_compute_using_mkl(); // Intel MKL functions use up to 4 threads on thread 0
    // and up to 12 threads on thread 1
}
my_compute_using_mkl(); // Intel MKL functions use up to 4 threads (!)
mkl_set_num_threads_local(0); // make master thread use global setting
my_compute_using_mkl(); // Intel MKL functions use up to 16 threads
```

This example shows how to avoid the side effect of a thread-local number of threads by saving and restoring the existing setting:

```c
#include "mkl.h"
void my_compute( int nt )
{
    int save = mkl_set_num_threads_local( nt ); // save the Intel® MKL number of threads
    my_compute_using_mkl(); // Intel MKL functions use up to nt threads on this thread
    mkl_set_num_threads_local( save ); // restore the Intel® MKL number of threads
}
```

**mkl_set_dynamic**

Enables Intel® MKL to dynamically change the number of OpenMP* threads.
Syntax
void mkl_set_dynamic (int flag);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| flag   | int  | flag = 0 - Requests disabling dynamic adjustment of the number of threads.  
|        |      | flag ≠ 0 - Requests enabling dynamic adjustment of the number of threads. |

Description
This function indicates whether Intel® MKL can dynamically change the number of OpenMP threads or should avoid doing this. The setting applies to all Intel® MKL functions on all execution threads. This function takes precedence over the MKL_DYNAMIC environment variable.

Dynamic adjustment of the number of threads is enabled by default. Specifically, Intel® MKL may use fewer threads in parallel regions than the number returned by the mkl_get_max_threads function. Disabling dynamic adjustment of the number of threads does not ensure that Intel® MKL actually uses the specified number of threads, although the library attempts to use that number.

Tip
If you call Intel® MKL from within an OpenMP parallel region and want to create internal parallel regions, either disable dynamic adjustment of the number of threads or set the thread-local number of threads (see mkl_set_num_threads_local for how to do it).

Example
```
#include "mkl.h"
...
mkl_set_num_threads( 8 );
#pragma omp parallel
{
    my_compute_with_mkl();  // Intel MKL uses 1 thread, being called from OpenMP parallel region
    mkl_set_dynamic( 0 );    // disable adjustment of the number of threads
    my_compute_with_mkl();   // Intel MKL uses 8 threads
}
```

mkl_get_max_threads

Gets the number of OpenMP* threads targeted for parallelism.

Syntax
int mkl_get_max_threads (void);
Include Files

- mkl.h

Description

This function returns the number of OpenMP threads available for Intel® MKL to use in internal parallel regions.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt</td>
<td>int</td>
<td>The maximum number of threads for Intel® MKL functions to use in internal parallel regions.</td>
</tr>
</tbody>
</table>

Example

```c
#include "mkl.h"
...
if (1 == mkl_get_max_threads()) puts("Intel MKL does not employ threading");
```

See Also

- mkl_set_dynamic
- mkl_get_dynamic
- mkl_domain_get_max_threads

**mkl_domain_get_max_threads**

*Gets the number of OpenMP* threads targeted for parallelism for a particular function domain.*

Syntax

```c
int mkl_domain_get_max_threads(int domain);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>domain</td>
<td>int</td>
<td>The named constant that defines the targeted domain.</td>
</tr>
</tbody>
</table>

Description

Computational functions of the Intel® MKL function domain defined by the `domain` parameter use the value returned by this function as a limit of the number of OpenMP threads they should request for parallel computations. The `mkl_domain_get_max_threads` function returns the thread-local number of threads or, if that value is zero or not set, the global number of threads. To determine this number, the function inspects the environment settings and return values of the function calls below in the order they are listed until it finds a non-zero value:

- A call to `mkl_set_num_threads_local`
- The last of the calls to `mkl_set_num_threads` or `mkl_domain_set_num_threads( ..., MKL_DOMAIN_ALL)`
- A call to `mkl_domain_set_num_threads( ..., domain)`
- The `MKL_DOMAIN_NUM_THREADS` environment variable with the `MKL_DOMAIN_ALL` tag
- The `MKL_DOMAIN_NUM_THREADS` environment variable (with the specific domain tag)
• The MKL_NUM_THREADS environment variable
• A call to omp_set_num_threads
• The OMP_NUM_THREADS environment variable

Actual number of threads used by the Intel® MKL computational functions may vary depending on the problem size and on whether dynamic adjustment of the number of threads is enabled (see the description of mkl_set_dynamic). For a list of supported values of the domain argument, see Table "Intel MKL Function Domains".

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt</td>
<td>int</td>
<td>The maximum number of threads for Intel® MKL functions from a given domain to use in internal parallel regions. If an invalid value of domain is supplied, the function returns the number of threads for MKL_DOMAIN_ALL</td>
</tr>
</tbody>
</table>

Example

```c
#include "mkl.h"
...
if (1 < mkl_domain_get_max_threads(MKL_DOMAIN_BLAS))
   puts("Intel MKL BLAS functions employ threading");
```

mkl_get_dynamic

*Determines whether Intel® MKL is enabled to dynamically change the number of OpenMP* threads.*

**Syntax**

```c
int mkl_get_dynamic(void);
```

**Include Files**

• mkl.h

**Description**

This function returns the status of dynamic adjustment of the number of OpenMP* threads. To determine this status, the function inspects the return value of the following function call and if it is undefined, inspects the environment setting below:

• A call to mkl_set_dynamic
• The MKL_DYNAMIC environment variable

**NOTE**

Dynamic adjustment of the number of threads is enabled by default.

The dynamic adjustment works as follows. Suppose that the mkl_get_max_threads function returns the number of threads equal to *N*. If dynamic adjustment is enabled, Intel® MKL may request up to *N* threads, depending on the size of the problem. If dynamic adjustment is disabled, Intel® MKL requests exactly *N* threads for internal parallel regions (provided it uses a threaded algorithm with at least *N* computations that can be done in parallel). However, the OpenMP* run-time library may be configured to supply fewer threads than Intel® MKL requests, depending on the OpenMP* setting of dynamic adjustment.
Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret</td>
<td>int</td>
<td>0 - Dynamic adjustment of the number of threads is disabled.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 - Dynamic adjustment of the number of threads is enabled.</td>
</tr>
</tbody>
</table>

Example

#include "mkl.h"
...
int nt = mkl_get_max_threads();
if (1 == mkl_get_dynamic())
printf("Intel MKL may use less than %i threads for a large problem", nt);
else
printf("Intel MKL should use %i threads for a large problem", nt);

mkl_set_num_stripes
Specifies the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.

Syntax
void mkl_set_num_stripes(int ns);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns</td>
<td>int</td>
<td>ns &gt; 0 - Specifies the number of partitions to use.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ns= 0 - Instructs Intel® MKL to use the default partitioning algorithm.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ns &lt; 0 - Invalid value; ignored.</td>
</tr>
</tbody>
</table>

Description
This function enables you to specify the number of stripes, or partitions along the leading dimension of the output matrix, for parallel ?gemm functions. If this number is not set (default) or if it is set to zero, Intel® MKL?gemm functions use the default partitioning algorithm. The specified number of partitions only applies to ?gemm functions.

The number specified is a hint, and Intel® MKL may actually use a smaller number.

NOTE
This function takes precedence over the MKL_NUM_STRIPES environment variable.
## mkl_get_num_stripes

*Gets the number of partitions along the leading dimension of the output matrix for parallel ?gemm functions.*

### Syntax

```c
int mkl_get_num_stripes(void);
```

### Include Files

- mkl.h

### Description

This function returns the number of stripes, that is, partitions along the leading dimension of the output matrix, for parallel ?gemm functions. The number of partitions only applies to ?gemm functions.

The number returned is a hint, and Intel® MKL may actually use a smaller number.

### Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns</td>
<td>int</td>
<td>The number of stripes for Intel® MKL?gemm functions to use.</td>
</tr>
</tbody>
</table>

### Example

```c
#include "mkl.h"
...
int ns = mkl_get_num_stripes();
if (ns > 0) printf("Intel MKL uses %d number of stripes\n", ns);
```

### See Also

mkl_set_num_stripes

---

## Error Handling

### Error Handling for Linear Algebra Routines

**xerbla**

*Error handling function called by BLAS, LAPACK, Vector Math, and Vector Statistics functions.*
Syntax

```c
void xerbla( const char * srname, const int* info, const int len );
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>srname</td>
<td>const char*</td>
<td>The name of the routine that called xerbla</td>
</tr>
<tr>
<td>info</td>
<td>const int*</td>
<td>The position of the invalid parameter in the parameter list of the calling function or an error code</td>
</tr>
<tr>
<td>len</td>
<td>const int</td>
<td>Length of the source string</td>
</tr>
</tbody>
</table>

Description

The `xerbla` function is an error handler for Intel® MKL BLAS, LAPACK, Vector Math, and Vector Statistics functions. These functions call `xerbla` if an issue is encountered on entry or during the function execution. `xerbla` operates as follows:

1. Prints a message that depends on the value of the `info` parameter as explained in the Error Messages Printed by `xerbla` table.

   **NOTE**
   
   A specific message can differ from the listed messages in numeric values and/or function names.

2. Returns to the calling application.

   Comments in the Netlib LAPACK reference code (http://www.netlib.org/lapack/explore-html/d1/dc0/b_l_a_s_2_s_r_c_2xerbla_8f.html) suggest this behavior although the LAPACK User's Guide recommends that the execution should stop when an error occurs.

Error Messages Printed by xerbla

<table>
<thead>
<tr>
<th>Value of info</th>
<th>Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Intel MKL ERROR: Incompatible optional parameters on entry to DGEMM.</td>
</tr>
<tr>
<td>1000 or 1089</td>
<td>Intel MKL INTERNAL ERROR: Insufficient workspace available in function CGELSD.</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Intel MKL INTERNAL ERROR: Condition 1 detected in function DLASD8.</td>
</tr>
<tr>
<td>Other</td>
<td>Intel MKL ERROR: Parameter 6 was incorrect on entry to DGEMM.</td>
</tr>
</tbody>
</table>

Note that `xerbla` is an internal function. You can change or disable printing of an error message by providing your own `xerbla` function. The following examples illustrate usage of `xerbla`.

Example

```c
void xerbla(char* srname, int* info, int len){
    // srname - name of the function that called xerbla
    // info - position of the invalid parameter in the parameter list
```

See Also
mkl_set_xerbla

pxerbla
Error handling routine called by ScaLAPACK routines.

Syntax
void pxerbla (MKL_INT* ictxt, char* srname, MKL_INT* info, MKL_INT srname_len);

Include Files
• mkl_scalapack.h

Input Parameters

ictxt  (local)
MKL_INT*
The BLACS context handle, indicating the global context of the operation.
The context itself is global.

srname  (global)
char*
The name of the routine that called pxerbla.

info  (global)
MKL_INT*
The position of the invalid parameter in the parameter list of the calling
routine.

srname_len  (global)
MKL_INT
The length of the calling routine name.

Description
This routine is an error handler for the ScaLAPACK routines. It is called if an input parameter has an invalid
value. A message is printed and program execution continues. For ScaLAPACK driver and computational
routines, a RETURN statement is issued following the call to pxerbla.

Control returns to the higher-level calling routine, and you can determine how the program should proceed.
However, in the specialized low-level ScaLAPACK routines (auxiliary routines that are Level 2 equivalents of
computational routines), the call to pxerbla() is immediately followed by a call to BLACS_ABORT() to
terminate program execution since recovery from an error at this level in the computation is not possible.

It is always good practice to check for a non-zero value of info on return from a ScaLAPACK routine.
Installers may consider modifying this routine in order to call system-specific exception-handling facilities.
LAPACKE_xerbla
Error handling function called by the C interface to LAPACK functions.

Syntax
void LAPACKE_xerbla(const char * name, lapack_int info);

Include Files
- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>const char*</td>
<td>The name of the routine that called LAPACKE_xerbla</td>
</tr>
<tr>
<td>info</td>
<td>lapack_int</td>
<td>The position of the invalid parameter in the parameter list of the calling function or an error code</td>
</tr>
</tbody>
</table>

Description
The LAPACKE_xerbla function is an error handler for Intel® MKL LAPACKE functions (the C interface to LAPACK functionality). If a LAPACKE function encounters an issue on entry or during the function execution, it calls LAPACKE_xerbla to print an error message and return an error code.

NOTE
The LAPACKE_xerbla routine does not replace the xerbla routine. For instance, if an issue occurs when a LAPACK function is called by a LAPACKE function, the LAPACK function calls xerbla.

Error Messages Printed by LAPACKE_xerbla

<table>
<thead>
<tr>
<th>Value of info</th>
<th>Example Error Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK_WORK_MEMORY_ERROR</td>
<td>Not enough memory to allocate work array in LAPACKE_dgees</td>
</tr>
<tr>
<td>LAPACK_TRANSPOSE_MEMORY_ERROR</td>
<td>Not enough memory to transpose matrix in LAPACKE_dgetrf_work</td>
</tr>
<tr>
<td>&lt; 0</td>
<td>Wrong parameter 1 in LAPACKE_dgetrf</td>
</tr>
</tbody>
</table>

NOTE
LAPACKE_xerbla is an internal function. You can change or disable printing of an error message by providing your own LAPACKE_xerbla function. Intel® MKL does not provide functionality for dynamic replacement of LAPACKE_xerbla.

See Also
xerbla
Handling Fatal Errors

A fatal error is a circumstance under which Intel® MKL cannot continue the computation. For example, a fatal error occurs when Intel® MKL cannot load a dynamic library or confronts an unsupported CPU type. In case of a fatal error, the default Intel® MKL behavior is to print an explanatory message to the console and call an internal function that terminates the application with a call to the system `exit()` function. Intel® MKL enables you to override this behavior by setting a custom handler of fatal errors. The custom error handler can be configured to throw a C++ exception, set a global variable indicating the failure, or otherwise handle cannot-continue situations. It is not necessary for the custom error handler to call the system `exit()` function. Once execution of the error handler completes, a call to Intel® MKL returns to the calling program without performing any computations and leaves no memory allocated by Intel® MKL and no thread synchronization pending on return.

To specify a custom fatal error handler, call the `mkl_set_exit_handler` function.

**mkl_set_exit_handler**

_SETS THE CUSTOM HANDLER OF FATAL ERRORS._

**Syntax**

```c
int mkl_set_exit_handler (MKLExitHandler myexit);
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Prototype</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>myexit</td>
<td><code>void (*myexit)(int why);</code></td>
<td>The error handler to set.</td>
</tr>
</tbody>
</table>

**Description**

This function sets the custom handler of fatal errors. If the input parameter is `NULL`, the system `exit()` function is set.

The following example shows how to use a custom handler of fatal errors in your C++ application:

```c
#include "mkl.h"

void my_exit(int why){
    throw my_exception();
}

int ComputationFunction()
{
    mkl_set_exit_handler( my_exit );
    try {
        compute_using_mkl();
    }
    catch (const my_exception& e) {
        handle_exception();
    }
}
```
Character Equality Testing

**lsame**
Tests two characters for equality regardless of the case.

**Syntax**

```c
int lsame( const char* ca, const char* cb, int lca, int lcb );
```

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca, cb</td>
<td>const char*</td>
<td>Pointers to the single characters to be compared</td>
</tr>
<tr>
<td>lca, lcb</td>
<td>int</td>
<td>Lengths of the input character strings, equal to one.</td>
</tr>
</tbody>
</table>

**Description**
This logical function checks whether two characters are equal regardless of the case.

**Return Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>int</td>
<td>Result of the comparison:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a non-zero value if ca is the same letter as cb, maybe except for the case.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• zero if ca and cb are different letters for whatever cases.</td>
</tr>
</tbody>
</table>

**lsamen**
Tests two character strings for equality regardless of the case.

**Syntax**

```c
MKL_INT lsamen( const MKL_INT* n, const char* ca, const char* cb );
```

**Include Files**
- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>const MKL_INT*</td>
<td>Pointer to the number of characters in ca and cb to be compared.</td>
</tr>
</tbody>
</table>
### Name

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca, cb</td>
<td>const char*</td>
<td>Character strings of length at least ( n ) to be compared. Only the first ( n ) characters of each string will be accessed.</td>
</tr>
</tbody>
</table>

### Description

This logical function tests whether the first \( n \) letters of one string are the same as the first \( n \) letters of the other string, regardless of the case.

### Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>MKL_INT</td>
<td>Result of the comparison:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• a non-zero value if the first ( n ) letters in ( ca ) and ( cb ) character strings are equal, maybe except for the case, or if the length of character string ( ca ) or ( cb ) is less than ( n ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• zero if the first ( n ) letters in ( ca ) and ( cb ) character strings are different for whatever cases.</td>
</tr>
</tbody>
</table>

### Timing

**second/dsecd**

*Returns elapsed time in seconds. Use to estimate real time between two calls to this function.*

**Syntax**

```c
float second( void );

double dsecd( void );
```

**Include Files**

- mkl.h

**Description**

The **second/dsecd** function returns time in seconds to be used to estimate real time between two calls to the function. The difference between these functions is in the precision of the floating-point type of the result: while **second** returns the single-precision type, **dsecd** returns the double-precision type.

Use these functions to measure durations. To do this, call each of these functions twice. For example, to measure performance of a routine, call the appropriate function directly before a call to the routine to be measured, and then after the call of the routine. The difference between the returned values shows real time spent in the routine.

Initializations may take some time when the **second/dsecd** function runs for the first time. To eliminate the effect of this extra time on your measurements, make the first call to **second/dsecd** in advance.

Do not use **second** to measure short time intervals because the single-precision format is not capable of holding sufficient timer precision.
Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>float</td>
<td>for second</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>for dsecnd</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Elapsed real time in seconds</td>
</tr>
</tbody>
</table>

mkl_get_cpu_clocks

*Returns elapsed CPU clocks.*

**Syntax**

```c
void mkl_get_cpu_clocks (unsigned MKL_INT64 *clocks);
```

**Include Files**

* mkl.h

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clocks</td>
<td>unsigned MKL_INT64</td>
<td>Elapsed CPU clocks</td>
</tr>
</tbody>
</table>

**Description**

The `mkl_get_cpu_clocks` function returns the elapsed CPU clocks. This may be useful when timing short intervals with high resolution. The `mkl_get_cpu_clocks` function is also applied in pairs like second/dsecnd. Note that out-of-order code execution on IA-32 or Intel® 64 architecture processors may disturb the exact elapsed CPU clocks value a little bit, which may be important while measuring extremely short time intervals.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

mkl_get_cpu_frequency

*Returns the current CPU frequency value in GHz.*

**Syntax**

```c
double mkl_get_cpu_frequency (void);
```

**Include Files**

* mkl.h
Description

The function `mkl_get_cpu_frequency` returns the current CPU frequency in GHz.

**NOTE**

The returned value may vary from run to run if power management or Intel® Turbo Boost Technology is enabled.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>double</td>
<td>Current CPU frequency value in GHz</td>
</tr>
</tbody>
</table>

**mkl_get_max_cpu_frequency**

*Returns the maximum CPU frequency value in GHz.*

**Syntax**

```c
double mkl_get_max_cpu_frequency(void);
```

**Include Files**

- `mkl.h`

**Description**

The function `mkl_get_max_cpu_frequency` returns the maximum CPU frequency in GHz.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freq</td>
<td>double</td>
<td>Maximum CPU frequency value in GHz</td>
</tr>
</tbody>
</table>

**mkl_get_clocks_frequency**

*Returns the frequency value in GHz based on constant-rate Time Stamp Counter.*

**Syntax**

```c
double mkl_get_clocks_frequency (void);
```

**Include Files**

- `mkl.h`

**Description**

The function `mkl_get_clocks_frequency` returns the CPU frequency value (in GHz) based on constant-rate Time Stamp Counter (TSC). Use of the constant-rate TSC ensures that each clock tick is constant even if the CPU frequency changes. Therefore, the returned frequency is constant.
Obtaining the frequency may take some time when `mkl_get_clocks_frequency` is called for the first time. The same holds for functions `second/dsecnd`, which call `mkl_get_clocks_frequency`.

### Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>freq</code></td>
<td>double</td>
<td>Frequency value in GHz</td>
</tr>
</tbody>
</table>

### See Also

- `second/dsecnd`

## Memory Management

This section describes the Intel® MKL memory functions. See the Intel® MKL Developer Guide for more memory usage information.

### mkl_free_buffers

*Frees unused memory allocated by the Intel® MKL Memory Allocator.*

#### Syntax

```c
void mkl_free_buffers (void);
```

#### Include Files

- `mkl.h`

#### Description

To improve performance of Intel® MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The `mkl_free_buffers` function frees unused memory allocated by the Memory Allocator.

See the Intel® MKL Developer Guide for details.

You should call `mkl_free_buffers` after the last call to Intel® MKL functions. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® MKL functions.

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
Usage of mkl_free_buffers with FFT Functions

```c
DFTI_DESCRIPTOR_HANDLE hand1;
DFTI_DESCRIPTOR_HANDLE hand2;
void mkl_free_buffers(void);

/* Using Intel MKL FFT */
Status = DftiCreateDescriptor(&hand1, DFTI_SINGLE, DFTI_COMPLEX, dim, m1);
Status = DftiCommitDescriptor(hand1);
Status = DftiComputeForward(hand1, s_array1);

Status = DftiCreateDescriptor(&hand2, DFTI_SINGLE, DFTI_COMPLEX, dim, m2);
Status = DftiCommitDescriptor(hand2);

Status = DftiFreeDescriptor(&hand1);
Status = DftiFreeDescriptor(&hand2);

/* Here you finish using Intel MKL FFT */
/* Memory leak will be triggered by any memory control tool */
/* Use mkl_free_buffers() to avoid memory leaking */

mkl_free_buffers();
```

mkl_thread_free_buffers

Frees unused memory allocated by the Intel® MKL Memory Allocator in the current thread.

**Syntax**

```c
void mkl_thread_free_buffers (void);
```

**Include Files**

- mkl.h

**Description**

To improve performance of Intel® MKL, the Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The mkl_thread_free_buffers function frees unused memory allocated by the Memory Allocator in the current thread only.

You should call mkl_thread_free_buffers after the last call to Intel® MKL functions in the current thread. In large applications, if you suspect that the memory may get insufficient, you may call this function earlier, but anticipate a drop in performance that may occur due to reallocation of buffers for subsequent calls to Intel® MKL functions.

**See Also**

mkl_free_buffers

mkl_disable_fast_mm

Turns off the Intel® MKL Memory Allocator for Intel® MKL functions to directly use the system malloc/free functions.

**Syntax**

```c
int mkl_disable_fast_mm (void);
```
Include Files

• mkl.h

Description

The `mkl_disable_fast_mm` function turns the Intel® MKL Memory Allocator off for Intel® MKL functions to directly use the system `malloc/free` functions. Intel® MKL Memory Allocator uses per-thread memory pools where buffers may be collected for fast reuse. The Memory Allocator is turned on by default for better performance. To turn it off, you can use the `mkl_disable_fast_mm` function or the `MKL_DISABLE_FAST_MM` environment variable. (See the Intel® MKL Developer Guide for details.) Call `mkl_disable_fast_mm` before calling any Intel® MKL functions that require allocation of memory buffers.

NOTE
Turning the Memory Allocator off negatively impacts performance of some Intel® MKL routines, especially for small problem sizes.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mm</td>
<td>int</td>
<td>1 - The Memory Allocator is successfully turned off. 0 - Turning the Memory Allocator off failed.</td>
</tr>
</tbody>
</table>

`mkl_mem_stat`

*Reports the status of the Intel® MKL Memory Allocator.*

Syntax

```c
MKL_INT64 mkl_mem_stat (int* AllocatedBuffers);
```

Include Files

• mkl.h

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocatedBuffers</td>
<td>int</td>
<td>The number of buffers allocated by Intel® MKL.</td>
</tr>
</tbody>
</table>

Description

The function returns the number of buffers allocated by Intel® MKL and the amount of memory in these buffers. Intel® MKL can allocate the memory buffers internally or in a call to `mkl_malloc/mkl calloc`. If no buffers are allocated at the moment, the `mkl_mem_stat` function returns 0. Call `mkl_mem_stat` to check the Intel® MKL memory status.

NOTE

If you free all the memory allocated in calls to `mkl_malloc` or `mkl calloc` and then call `mkl_freeBuffers`, a subsequent call to `mkl_mem_stat` normally returns 0.
Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocatedBytes</td>
<td>MKL_INT64</td>
<td>The amount of allocated memory (in bytes).</td>
</tr>
</tbody>
</table>

See Also
Usage Example for the Memory Functions

**mkl_peak_mem_usage**

Reports the peak memory allocated by the Intel® MKL Memory Allocator.

**Syntax**

```c
MKL_INT64 mkl_peak_mem_usage (int mode);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>int</td>
<td>Requested mode of the function's operation. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MKL_PEAK_MEM_ENABLE - start gathering the peak memory data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MKL_PEAK_MEM_DISABLE - stop gathering the peak memory data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MKL_PEAK_MEM - return the peak memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• MKL_PEAK_MEM_RESET - return the peak memory and reset the counter to start gathering the peak memory data from scratch</td>
</tr>
</tbody>
</table>

**Description**

The `mkl_peak_mem_usage` function reports the peak memory allocated by the Intel® MKL Memory Allocator. Gathering the peak memory data is turned off by default. If you need to know the peak memory, explicitly turn the data gathering mode on by calling the function with the MKL_PEAK_MEM_ENABLE value of the parameter. Use the MKL_PEAK_MEM and MKL_PEAK_MEM_RESET values only when the data gathering mode is turned on. Otherwise the function returns -1. The data gathering mode leads to performance degradation, so when the mode is turned on, you can turn it off by calling the function with the MKL_PEAK_MEM_DISABLE value of the parameter.

**NOTE**

- If Intel® MKL is running in a threaded mode, the `mkl_peak_mem_usage` function may return different amounts of memory from run to run.
- The function reports the peak memory for the entire application, not just for the calling thread.
### Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllocatedBytes</td>
<td>MKL_INT64</td>
<td>The peak memory allocated by the Memory Allocator (in bytes) or -1 in case of errors.</td>
</tr>
</tbody>
</table>

### See Also

Usage Example for the Memory Functions

#### mkl_malloc

*Allocates an aligned memory buffer.*

#### Syntax

```c
void* mkl_malloc (size_t alloc_size, int alignment);
```

#### Include Files

- mkl.h

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>alloc_size</td>
<td>size_t</td>
<td>Size of the buffer to be allocated.</td>
</tr>
<tr>
<td>alignment</td>
<td>int</td>
<td>Alignment of the buffer.</td>
</tr>
</tbody>
</table>

#### Description

The function allocates an `alloc_size`-byte buffer aligned on the `alignment`-byte boundary. If `alignment` is not a power of 2, the 32-byte alignment is used.

#### Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_ptr</td>
<td>void*</td>
<td>Pointer to the allocated buffer if <code>alloc_size</code> ≥ 1, NULL if <code>alloc_size</code> &lt; 1.</td>
</tr>
</tbody>
</table>

### See Also

- mkl_free

Usage Example for the Memory Functions

#### mklcalloc

*Allocates and initializes an aligned memory buffer.*

#### Syntax

```c
void* mkl_calloc (size_t num, size_t size, int alignment);
```

#### Include Files

- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>size_t</td>
<td>The number of elements in the buffer to be allocated.</td>
</tr>
<tr>
<td>size</td>
<td>size_t</td>
<td>The size of the element.</td>
</tr>
<tr>
<td>alignment</td>
<td>int</td>
<td>Alignment of the buffer.</td>
</tr>
</tbody>
</table>

Description

The function allocates a \( num \times size \)-byte buffer, aligned on the \( alignment \)-byte boundary, and initializes the buffer with zeros.

If \( alignment \) is not a power of 2, the 64-byte alignment is used.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_ptr</td>
<td>void*</td>
<td>Pointer to the allocated buffer if ( size \geq 1 ), NULL if ( size &lt; 1 ).</td>
</tr>
</tbody>
</table>

See Also

- mkl_malloc
- mkl_realloc
- mkl_free

Usage Example for the Memory Functions

**mkl_realloc**

Changes the size of memory buffer allocated by mkl_malloc/mkl_calloc.

**Syntax**

```c
void* mkl_realloc (void *ptr, size_t size);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>void*</td>
<td>Pointer to the memory buffer allocated by the mkl_malloc or mkl_calloc function or a NULL pointer.</td>
</tr>
<tr>
<td>size</td>
<td>size_t</td>
<td>New size of the buffer.</td>
</tr>
</tbody>
</table>
Description

The function changes the size of the memory buffer allocated by the `mkl_malloc` or `mklcalloc` function to `size` bytes. The first bytes of the returned buffer up to the minimum of the old and new sizes keep the content of the input buffer. The returned memory buffer can have a different location than the input one. If `ptr` is `NULL`, the function works as `mkl_malloc`.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| `a_ptr` | `void*` | Pointer to the re-allocated buffer if re-allocation is successful.  
|        |      | NULL if re-allocation is unsuccessful. |

See Also

- `mkl_malloc`
- `mklcalloc`
- `mkl_free`

Usage Example for the Memory Functions

**mkl_free**

*Frees the aligned memory buffer allocated by mkl_malloc/mklcalloc.*

Syntax

```c
void mkl_free (void *a_ptr);
```

Include Files

- `mkl.h`

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a_ptr</code></td>
<td><code>void*</code></td>
<td>Pointer to the buffer to be freed.</td>
</tr>
</tbody>
</table>

Description

The function frees the buffer pointed by `a_ptr` and allocated by the `mkl_malloc()` or `mklcalloc()` function and does nothing if `a_ptr` is `NULL`.

See Also

- `mkl_malloc`
- `mklcalloc`

Usage Example for the Memory Functions

**mkl_set_memory_limit**

*On Linux, sets the limit of memory that Intel® MKL can allocate for a specified type of memory.*
Syntax

```c
int mkl_set_memory_limit (int mem_type, size_t limit);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_type</td>
<td>int</td>
<td>Type of memory to limit. Possible values:</td>
</tr>
<tr>
<td>limit</td>
<td>size_t</td>
<td>Memory limit in megabytes.</td>
</tr>
</tbody>
</table>

Description

This function sets the limit for the amount of memory that Intel® MKL can allocate for the specified memory type. The limit bounds both internal allocations (inside Intel® MKL computation routines) and external allocations (in a call to mkl_malloc, mkl_calloc, or mkl_realloc). By default no limit is set for memory allocation.

Call `mkl_set_memory_limit` at most once, prior to calling any other Intel® MKL function in your application except `mkl_set_interface_layer` and `mkl_set_threading_layer`.

**NOTE**

- Allocation in MCDRAM requires `libmemkind` and `libjemalloc` dynamic libraries which are a part of Intel® Manycore Platform Software Package (Intel® MPSP) for Linux*.
- The `mkl_set_memory_limit` function takes precedence over the MKL_FAST_MEMORY_LIMIT environment variable.

Return Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Status of the function completion:</td>
</tr>
<tr>
<td></td>
<td>• 1 - the limit is set</td>
</tr>
<tr>
<td></td>
<td>• 0 - the limit is not set</td>
</tr>
</tbody>
</table>

See Also

- mkl_malloc
- mkl_calloc
- mkl_realloc

Usage Example for the Memory Functions
**Usage Example for the Memory Functions**

```c
#include <stdio.h>
#include <mkl.h>

int main(void) {
    double *a, *b, *c;
    int n, i;
    double alpha, beta;
    MKL_INT64 AllocatedBytes;
    int N_AllocatedBuffers;

    alpha = 1.1; beta = -1.2;
    n = 1000;
    mkl_peak_mem_usage(MKL_PEAK_MEM_ENABLE);
    a = (double*)mkl_malloc(n*n*sizeof(double),64);
    b = (double*)mkl_malloc(n*n*sizeof(double),64);
    c = (double*)mkl_malloc(n*n*sizeof(double),64);
    for (i=0;i<(n*n);i++) {
        a[i] = (double)(i+1);
        b[i] = (double)(-i-1);
    }
    dgemm("N","N",&n,&n,&n,&alpha,a,&n,b,&n,&beta,c,&n);
    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    printf("DGEMM uses %d bytes in %d buffers",AllocatedBytes,N_AllocatedBuffers);
    mkl_free_buffers();
    mkl_free(a);
    mkl_free(b);
    mkl_free(c);
    AllocatedBytes = mkl_mem_stat(&N_AllocatedBuffers);
    if (AllocatedBytes > 0) {
        printf("MKL memory leak!");
        printf("After mkl_free_buffers there are %d bytes in %d buffers",AllocatedBytes,N_AllocatedBuffers);
    }
    printf("Peak memory allocated by Intel MKL memory allocator %d bytes. Start to count new memory peak",
           mkl_peak_mem_usage(MKL_PEAK_MEM_RESET));
    a = (double*)mkl_malloc(n*n*sizeof(double),64);
    a = (double*)mkl_realloc(a,2*n*n*sizeof(double));
    mkl_free(a);
    printf("Peak memory allocated by Intel MKL memory allocator after reset of peak memory counter %d bytes\n",
           mkl_peak_mem_usage(MKL_PEAK_MEM));
    return 0;
}
```

---

**Single Dynamic Library Control**

Intel® MKL provides the Single Dynamic Library (SDL), which enables setting the interface and threading layer for Intel® MKL at runtime. See Intel® MKL Developer Guide for details of SDL and layered model concept. This section describes the functions supporting SDL.
**mkl_set_interface_layer**

Sets the interface layer for Intel® MKL at run time. Use with the Single Dynamic Library.

**Syntax**

```c
int mkl_set_interface_layer (int required_interface);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required_interface</td>
<td>int</td>
<td>Determines the interface layer. Possible values depend on the system architecture. Some of the values are only available on Linux* OS:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Intel® 64 architecture:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_LP64 for the Intel LP64 interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_ILP64 for the Intel ILP64 interface.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_LP64+MKL_INTERFACE_GNU for the GNU* LP64 interface on Linux OS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_ILP64+MKL_INTERFACE_GNU for the GNU ILP64 interface on Linux OS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- IA-32 architecture:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_LP64 for the Intel interface on Linux OS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- MKL_INTERFACE_LP64+MKL_INTERFACE_GNU or MKL_INTERFACE_GNU for the GNU interface on Linux OS.</td>
</tr>
</tbody>
</table>

**Description**

If you are using the Single Dynamic Library (SDL), the `mkl_set_interface_layer` function sets the specified interface layer for Intel® MKL at run time.

Call this function prior to calling any other Intel® MKL function in your application except `mkl_set_threading_layer`. You can call `mkl_set_interface_layer` and `mkl_set_threading_layer` in any order.

The `mkl_set_interface_layer` function takes precedence over the `MKL_INTERFACE_LAYER` environment variable.

See *Intel® MKL Developer Guide* for the layered model concept and usage details of the SDL.

**Return Values**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>- Current interface layer if it is set in a call to <code>mkl_set_interface_layer</code> or specified by environment variables or defaults.</td>
</tr>
</tbody>
</table>
**mkl_set_threading_layer**

Sets the threading layer for Intel® MKL at run time.

*Use with the Single Dynamic Library (SDL).*

**Syntax**

```c
int mkl_set_threading_layer (int required_threading);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>required_threading</td>
<td>int</td>
<td>Determines the threading layer. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MKL_THREADING_INTEL for Intel threading,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MKL_THREADING_SEQUENTIAL for the sequential mode of Intel® MKL.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MKL_THREADING_TBB for threading with the Intel® Threading Building Blocks.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MKL_THREADING_PGI for PGI threading on Windows* or Linux* operating system only. Do not use this value with the SDL for Intel® Many Integrated Core (Intel® MIC) Architecture.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MKL_THREADING_GNU for GNU threading on Linux* operating system only. Do not use this value with the SDL for Intel MIC Architecture.</td>
</tr>
</tbody>
</table>

**Description**

If you are using the Single Dynamic Library (SDL), the `mkl_set_threading_layer` function sets the specified threading layer for Intel® MKL at run time.

Call this function prior to calling any other Intel® MKL function in your application except `mkl_set_interface_layer`.

You can call `mkl_set_threading_layer` and `mkl_set_interface_layer` in any order.

The `mkl_set_threading_layer` function takes precedence over the MKL_THREADING_LAYER environment variable.

See Intel® MKL Developer Guide for the layered model concept and usage details of the SDL.
Return Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| int    | • Current threading layer if it is set in a call to mkl_set_threading_layer or specified by environment variables or defaults. Possible values are specified in Input Parameters.  
• -1, if the layer was not specified prior to the call and the input parameter is incorrect. |

mkl_set_xerbla
Replaces the error handling routine. Use with the Single Dynamic Library.

Syntax
XerblaEntry mkl_set_xerbla (XerblaEntry new_xerbla_ptr);

Include Files
• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_xerbla_ptr</td>
<td>XerblaEntry</td>
<td>Pointer to the error handling routine to be used.</td>
</tr>
</tbody>
</table>

Description
The mkl_set_xerbla function replaces the error handling routine that is called by Intel® MKL functions with the routine specified by the parameter.
See Intel® MKL Developer Guide for details about SDL.

Return Values
The function returns the pointer to the replaced error handling routine.

See Also
xerbla

mkl_set_progress
Replaces the progress information routine.

Syntax
ProgressEntry mkl_set_progress (ProgressEntry new_progress_ptr);

Include Files
• mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_progress_ptr</td>
<td>ProgressEntry</td>
<td>Pointer to the progress information routine to be used.</td>
</tr>
</tbody>
</table>

Description

The `mkl_set_progress` function replaces the currently used progress information routine with the routine specified by the parameter.

Usually a user-supplied `mkl_progress` function redefines the default `mkl_progress` function automatically. However, you must call `mkl_set_progress` to replace the default `mkl_progress` on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) `mkl_rt.lib`.
- You link dynamically with ScaLAPACK.

See *Intel® MKL Developer Guide* for details of SDL.

Return Values

The function returns the pointer to the replaced progress information routine.

See Also

`mkl_progress`

mkl_set_pardiso_pivot

*Replaces the routine handling Intel® MKL PARDISO pivots with a user-defined routine. Use with the Single Dynamic Library (SDL).*

Syntax

```c
PardisopivotEntry mkl_set_pardiso_pivot (PardisopivotEntry new_pardiso_pivot_ptr);
```

Include Files

- `mkl.h`

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>new_pardiso_pivot_ptr</td>
<td>PardisopivotEntry</td>
<td>Pointer to the pivot setting routine to be used.</td>
</tr>
</tbody>
</table>

Description

If you are using the Single Dynamic Library (SDL), the `mkl_set_pardiso_pivot` function replaces the pivot setting routine that is called by Intel® MKL functions with the routine specified by the parameter.

See *Intel® MKL Developer Guide* for usage details of the SDL.
Return Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PardisopivotEntry</td>
<td>Pointer to the replaced pivot setting routine.</td>
</tr>
</tbody>
</table>

See Also

mkl_pardiso_pivot

Conditional Numerical Reproducibility Control

The CNR mode of Intel® MKL ensures bitwise reproducible results from run to run of Intel® MKL functions on a fixed number of threads for a specific Intel instruction set architecture (ISA) under the following conditions:

- Calls to Intel® MKL occur in a single executable
- The number of computational threads used by the library does not change in the run

Intel® MKL offers both functions and environment variables to support conditional numerical reproducibility. See the Intel® MKL Developer Guide for more information on bitwise reproducible results of computations and for details about the environment variables.

The support functions enable you to configure the CNR mode and also provide information on the current and optimal CNR branch on your system. Usage Examples for CNR Support Functions illustrate usage of these functions.

Important

Call the functions that define the behavior of CNR before any of the math library functions that they control.

Intel® MKL provides named constants for use as input and output parameters of the functions instead of integer values. See Named Constants for CNR Control for a list of the named constants.

Although you can configure the CNR mode using either the support functions or the environment variables, the functions offer more flexible configuration and control than the environment variables. Settings specified by the functions take precedence over the settings specified by the environment variables.

Use Intel® MKL in the CNR mode only in case a need for bitwise reproducible results is critical. Otherwise, run Intel® MKL as usual to avoid performance degradation.

While you can supply unaligned input and output data to Intel® MKL functions running in the CNR mode, use of aligned data is recommended. Refer to Reproducibility Conditions for more details.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804
mkl_cbwr_set
Configures the CNR mode of Intel® MKL.

Syntax

```c
int mkl_cbwr_set (int setting);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setting</td>
<td>int</td>
<td>CNR branch to set. See Named Constants for CNR Control for a list of named constants that specify the settings.</td>
</tr>
</tbody>
</table>

Description

The `mkl_cbwr_set` function configures the CNR mode. In this release, it sets the CNR branch and turns on the CNR mode.

**NOTE**
Settings specified by the `mkl_cbwr_set` function take precedence over the settings specified by the `MKL_CBWR` environment variable.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| status | int  | The status of the function completion:

- `MKL_CBWR_SUCCESS` - the function completed successfully.
- `MKL_CBWR_ERR_INVALID_INPUT` - an invalid setting is requested.
- `MKL_CBWR_ERR_UNSUPPORTED_BRANCH` - the input value of the branch does not match the instruction set architecture (ISA) of your system. See Named Constants for CNR Control for more details.
- `MKL_CBWR_ERR_MODE_CHANGE_FAILURE` - the `mkl_cbwr_set` function requested to change the current CNR branch after a call to some Intel® MKL function other than a CNR function.

See Also

Usage Examples for CNR Support Functions

mkl_cbwr_get
Returns the current CNR settings.
Syntax

```c
int mkl_cbwr_get (int option);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| option  | int    | Specifies the CNR settings requested. Named constants define possible values of `option`:
    |        | • `MKL_CBWR_BRANCH` - returns the current CNR branch only.                   |
    |        | • `MKL_CBWR_ALL` - returns all CNR settings including strict CNR setting.    |

Description

The `mkl_cbwr_get` function returns the requested CNR settings. The function returns `MKL_CBWR_ERR_INVALID_INPUT` if an invalid option is specified.

**NOTE**

To enable CNR mode, use the `mkl_cbwr_set` function or environment variables. For more details, see the *Intel® MKL Developer Guide*.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setting</td>
<td>int</td>
<td>Requested CNR settings. See <a href="#">Named Constants for CNR Control</a> for a list of named constants that specify the settings.</td>
</tr>
</tbody>
</table>
<pre><code>|        | If the value of the `option` parameter is not permitted, contains the `MKL_CBWR_ERR_INVALID_INPUT` error code. |
</code></pre>

See Also

Usage Examples for CNR Support Functions

- `mkl_cbwr_set`

`mkl_cbwr_get_auto_branch`

*Automatically detects the CNR code branch for your platform.*

Syntax

```c
int mkl_cbwr_get_auto_branch (void);
```

Include Files

- mkl.h
Description

The `mkl_cbwr_get_auto_branch` function uses a run-time CPU check to return a CNR branch that is optimized for the processor where the program is currently running.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setting</code></td>
<td>int</td>
<td>Automatically detected CNR branch. May be any specific branch listed in Named Constants for CNR Control.</td>
</tr>
</tbody>
</table>

See Also

Usage Examples for CNR Support Functions

Named Constants for CNR Control

Use the conditional numerical reproducibility (CNR) functionality in Intel® MKL to obtain reproducible results from MKL routines. When enabling CNR, you choose a specific code branch of Intel® MKL that corresponds to the instruction set architecture (ISA) that you target. Use these named constants to specify the code branch and other CNR options.

<table>
<thead>
<tr>
<th>Named Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNR Branches</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MKL_CBWR_OFF</td>
<td>0</td>
<td>Disable CNR mode</td>
</tr>
<tr>
<td>MKL_CBWR.Branch_Off</td>
<td>1</td>
<td>CNR mode is disabled</td>
</tr>
<tr>
<td>MKL_CBWR_AUTO</td>
<td>2</td>
<td>Choose branch automatically. CNR mode uses the standard ISA-based dispatching model while ensuring fixed cache sizes, deterministic reductions, and static scheduling</td>
</tr>
<tr>
<td>MKL_CBWR_COMPATIBLE</td>
<td>3</td>
<td>Intel® Streaming SIMD Extensions 2 (Intel® SSE2) without rcpps/rsqrtps instructions</td>
</tr>
<tr>
<td>MKL_CBWR_SSE2</td>
<td>4</td>
<td>Intel SSE2</td>
</tr>
<tr>
<td>MKL_CBWR_SSE3</td>
<td>5</td>
<td>DEPRECATED. Intel® Streaming SIMD Extensions 3 (Intel® SSE3). This setting is kept for backward compatibility and is equivalent to MKL_CBWR_SSE2.</td>
</tr>
<tr>
<td>MKL_CBWR_SSSE3</td>
<td>6</td>
<td>Supplemental Streaming SIMD Extensions 3 (SSSE3)</td>
</tr>
</tbody>
</table>
**Named Constant** | **Value** | **Description**
---|---|---
MKL_CBWR_SSE4_1 | 7 | Intel® Streaming SIMD Extensions 4-1 (SSE4-1)
MKL_CBWR_SSE4_2 | 8 | Intel® Streaming SIMD Extensions 4-2 (SSE4-2)
MKL_CBWR_AVX | 9 | Intel® Advanced Vector Extensions (Intel® AVX)
MKL_CBWR_AVX2 | 10 | Intel® Advanced Vector Extensions 2 (Intel® AVX2)
MKL_CBWR_AVX512_MIC | 11 | Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon Phi™ processors
MKL_CBWR_AVX512 | 12 | Intel AVX-512 on Intel® Xeon® processors
MKL_CBWR_AVX512_MIC_E1 | 13 | Intel® Advanced Vector Extensions 512 (Intel® AVX-512) for Intel® Many Integrated Core Architecture (Intel® MIC Architecture) with support of AVX512_4FMAPS and AVX512_4VNNIW instruction groups enabled processors
MKL_CBWR_AVX512_E1 | 14 | Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support of Vector Neural Network Instructions enabled processors

**CNR Flags**

**MKL_CBWR_STRICT** | 64 | Strict CNR mode enabled. See Reproducibility Conditions for more information.

When specifying the CNR branch with the named constants, be aware of the following:

- Reproducible results are provided under Reproducibility Conditions.
- Settings other than MKL_CBWR_AUTO or MKL_CBWR_COMPATIBLE are available only for Intel processors.
- Intel and Intel compatible CPUs have a few instructions, such as approximation instructions rcpps/rsqrtps, that may return different results. Setting the branch to MKL_CBWR_COMPATIBLE ensures that Intel® MKL does not use these instructions and forces a single Intel SSE2-only code path to be executed.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**See Also**

Usage Examples for CNR Support Functions

**Reproducibility Conditions**

To get reproducible results from run to run, ensure that the number of threads is fixed and constant. Specifically:

- If you are running your program with OpenMP* parallelization on different processors, explicitly specify the number of threads.
• To ensure that your application has deterministic behavior with OpenMP* parallelization and does not adjust the number of threads dynamically at run time, set MKL_DYNAMIC and OMP_DYNAMIC to FALSE. This is especially needed if you are running your program on different systems.
• If you are running your program with the Intel® Threading Building Blocks parallelization, numerical reproducibility is not guaranteed.

Strict CNR Mode
In strict CNR mode, Intel® MKL provides bitwise reproducible results for a limited set of functions and code branches even when the number of threads changes. These routines and branches support strict CNR mode (64-bit libraries only):

- Intel® Advanced Vector Extensions 2 (Intel® AVX2) or Intel® Advanced Vector Extensions 512 (Intel® AVX-512).

When using other routines or CNR branches, Intel® MKL operates in standard (non-strict) CNR mode, subject to the restrictions described above. Enabling strict CNR mode can reduce performance.

NOTE
- As usual, you should align your data, even in CNR mode, to obtain the best possible performance. While CNR mode also fully supports unaligned input and output data, the use of it might reduce the performance of some Intel® MKL functions on earlier Intel processors. To ensure proper alignment of arrays, allocate memory for them using mkl_malloc/mkl_calloc.
- Conditional Numerical Reproducibility does not ensure that bitwise-identical NaN values are generated when the input data contains NaN values.
- If dynamic memory allocation fails on one run but succeeds on another run, you may fail to get reproducible results between these two runs.

See Also
mkl_malloc
mkl_calloc

Usage Examples for CNR Support Functions
The following examples illustrate usage of support functions for conditional numerical reproducibility.

Setting Automatically Detected CNR Branch
```c
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Find the available MKL_CBWR_BRANCH automatically */
    my_cbwr_branch = mkl_cbwr_get_auto_branch();
    /* User code without Intel MKL calls */
    /* Piece of the code where CNR of Intel MKL is needed */
    /* The performance of Intel MKL functions might be reduced for CNR mode */
    if (mkl_cbwr_set(my_cbwr_branch)!=MKL_CBWR_SUCCESS) {
        printf("Error in setting MKL_CBWR_BRANCH! Aborting\n");
        return;
    }
    /* CNR calls to Intel MKL + any other code */
}
```
**Use of the mkl_cbwr_get Function**

```c
#include <mkl.h>
int main(void) {
    int my_cbwr_branch;
    /* Piece of the code where CNR of Intel MKL is analyzed */
    my_cbwr_branch = mkl_cbwr_get(MKL_CBWR_BRANCH);
    switch (my_cbwr_branch) {
        case MKL_CBWR_AUTO:
            /* actions in case of automatic mode */
            break;
        case MKL_CBWR_SSSE3:
            /* actions for SSSE3 code */
            break;
        default:
            /* all other cases */
    }
    /* User code */
}
```

**Miscellaneous**

**mkl_progress**

*Provides progress information.*

**Syntax**

```c
int mkl_progress (int* thread_process, int* step, char* stage, int lstage);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_process</td>
<td>const int*</td>
<td>Indicates the number of thread or process the progress routine is called from:</td>
</tr>
<tr>
<td>step</td>
<td>const int*</td>
<td>Pointer to the linear progress indicator that shows the amount of work done. Increases from 0 to the linear size of the problem during the computation.</td>
</tr>
<tr>
<td>stage</td>
<td>const char*</td>
<td>Message indicating the name of the routine or the name of the computation stage the progress routine is called from.</td>
</tr>
<tr>
<td>lstage</td>
<td>int</td>
<td>The length of a stage string excluding the trailing NULL character.</td>
</tr>
</tbody>
</table>
Description

The mkl_progress function is intended to track progress of a lengthy computation and/or interrupt the computation. By default this routine does nothing but the user application can redefine it to obtain the computation progress information. You can set it to perform certain operations during the routine computation, for instance, to print a progress indicator. A non-zero return value may be supplied by the redefined function to break the computation.

Some Intel® MKL functions from LAPACK, ScALAPACK, DSS/PARDISO, and Parallel Direct Sparse Solver for Clusters regularly call the mkl_progress function during the computation. Refer to the description of a specific function from those domains to see whether the function supports this feature or not.

If a LAPACK function returns info=-1002, the function was interrupted by mkl_progress. Because ScALAPACK does not support interruption of the computation, Intel® MKL ignores any value returned by mkl_progress.

While a user-supplied mkl_progress function usually redefines the default mkl_progress function automatically, some configurations require calling the mkl_set_progress function to replace the default mkl_progress function. Call mkl_set_progress to replace the default mkl_progress on Windows* in any of the following cases:

- You are using the Single Dynamic Library (SDL) mkl_rt.lib.
- You link dynamically with ScALAPACK.

Warning

The mkl_progress function only supports OpenMP* threading and sequential execution.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stopflag</td>
<td>int</td>
<td>The stopping flag. A non-zero flag forces the routine to be interrupted. The zero flag is the default return value.</td>
</tr>
</tbody>
</table>

Example

The following example prints the progress information to the standard output device:

```c
#include <stdio.h>
#include <string.h>
#define BUFLEN 16
int mkl_progress( int* thread_process, int* step, char* stage, int lstage )
{
    char buf[BUFLEN];
    if( lstage >= BUFLEN ) lstage = BUFLEN-1;
    strncpy( buf, stage, lstage );
    buf[lstage] = '\0';
    printf( "In thread %i, at stage %s, steps passed %i\n", *thread_process, buf, *step );
    return 0;
}
```

mkl_enable_instructions

Enables dispatching for new Intel® architectures or restricts the set of Intel® instruction sets available for dispatching.
Syntax

```c
int mkl_enable_instructions (int isa);
```

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isa</td>
<td>int</td>
<td>The latest Intel® instruction-set architecture (ISA) for Intel® MKL to dispatch.</td>
</tr>
</tbody>
</table>

- **MKL_ENABLE_AVX512**: Intel® Advanced Vector Extensions 512 (Intel® AVX-512) on Intel® Xeon® processors.
- **MKL_ENABLE_AVX512_E1**: Intel® Advanced Vector Extensions 512 (Intel® AVX-512) with support for Vector Neural Network Instructions.
- **MKL_ENABLE_AVX512_E2**: ICX: Intel® Advanced Vector Extensions 512 (Intel® AVX-512) enabled processors.
- **MKL_ENABLE_AVX512_MIC**: Intel AVX-512 on Intel® Xeon Phi™ processors.
- **MKL_ENABLE_AVX2**: Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- **MKL_ENABLE_AVX**: Intel® Advanced Vector Extensions (Intel® AVX).
- **MKL_ENABLE_SSE4_2**: Intel® Streaming SIMD Extensions 4-2 (Intel® SSE4-2).

Description

Intel® MKL does run-time processor dispatching to identify appropriate internal code paths to traverse for Intel® MKL functions called by the application. The `mkl_enable_instructions` function controls the behavior of the dispatcher to do either of the following:

- Enable dispatching for new Intel architectures.
  
  Intel® MKL does not dispatch instruction sets that do not have silicon available at time of the product launch. Call `mkl_enable_instructions` to enable dispatching the code path for such an ISA in a simulator environment or on hardware that supports this ISA.

- Restrict the set of Intel instruction sets available for dispatching.
  
  Call `mkl_enable_instructions` to restrict dispatching to code paths for earlier ISA. For example, if the hardware supports Intel AVX, a call to `mkl_enable_instructions` with the `MKL_ENABLE_SSE4_2` parameter forces the dispatcher to use the Intel SSE4-2 code path.

If the system does not support the instruction set specified by the `isa` parameter or if the system is based on a non-Intel architecture, `mkl_enable_instructions` does nothing and returns zero.
Settings specified by the `mkl_enable_instructions` function set an upper limit to settings specified by the `mkl_cbwr_set` function.

You can use the `MKL_ENABLE_INSTRUCTIONS` environment variable instead of calling `mkl_enable_instructions` (for more details, see the Intel® MKL Developer Guide); however, the settings specified by the function take precedence over the settings specified by the environment variable.

**Return Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| irc  | int  | Function completion status:  
1 - Intel® MKL dispatches the code path for the specified ISA by default.  
0 - The request is rejected. Usually this occurs if `mkl_enable_instructions` was called:  
• After another Intel® MKL function  
• On a non-Intel architecture  
• With an incompatible ISA specified |

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**mkl_set_env_mode**

Sets up the mode that ignores environment settings specific to Intel® MKL.

**Syntax**

```c
int mkl_set_env_mode(int mode);
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
</table>
| mode | int  | Specifies what mode to set. For details, see Description. Possible values:  
• 0 - Do nothing.  
Use this value to query the current environment mode. |
1 - Make Intel® MKL ignore environment settings specific to the library.

**Description**

In the default *environment mode*, Intel® MKL can control its behavior using environment variables for threading, memory management, Conditional Numerical Reproducibility, automatic offload, and so on. The `mkl_set_env_mode` function sets up the environment mode that ignores all settings specified by Intel® MKL environment variables except `MIC_LD_LIBRARY_PATH` and `MKLROOT`.

**Return Values**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>current_mode</code></td>
<td>int</td>
<td>Environment mode that was used before the function call:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 - Default</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 - Ignore environment settings specific to Intel® MKL.</td>
</tr>
</tbody>
</table>

**mkl_verbose**

*Enables or disables Intel® MKL Verbose mode.*

**Syntax**

```c
int mkl_verbose (int enable);
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>enable</code></td>
<td>int</td>
<td>Desired state of the Intel® MKL Verbose mode. Indicates whether printing Intel® MKL function call information should be turned on or off. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 - disable the Verbose mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 - enable the Verbose mode.</td>
</tr>
</tbody>
</table>

**Description**

This function enables or disables the Intel® MKL Verbose mode, in which computational functions print call description information. For details of the Verbose mode, see the *Intel® MKL Developer Guide*, available in the Intel® Software Documentation Library.

**NOTE**

The setting for the Verbose mode specified by the `mkl_verbose` function takes precedence over the setting specified by the `MKL_VERBOSE` environment variable.
Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

- If the requested operation completed successfully, contains previous state of the verbose mode:
  - 0 - disabled
  - 1 - enabled
- If the function failed to complete the operation because of an incorrect input parameter, equals -1.

See Also
Intel Software Documentation Library

mklVerboseoutput_file
Write output in Intel® MKL Verbose mode to a file.

Syntax

```c
int mkl_verbose_output_file (const char* filename);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>char</td>
<td>Name of file. Specify the complete path of the output file.</td>
</tr>
</tbody>
</table>

Description

This function writes the output in Verbose mode to the file specified in the path.
If the write operation is successful, the function returns 0.
If the file does not exist or cannot be opened, the write operation is unsuccessful. The function returns 1 and defaults to mkl_verbose behavior by printing to stdout.
For more information on the Verbose mode, see the Intel® MKL Developer Guide, available in the Intel® Software Documentation Library.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>

- 0 indicates that the write operation was successful.
- 1 indicates that the write operation was unsuccessful.

See Also
Intel Software Documentation Library

mkl_set_mpi
Sets the implementation of the message-passing interface to be used by Intel® MKL.
Syntax

```c
int mkl_set_mpi (int vendor, const char *custom_library_name);
```

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vendor</td>
<td>int</td>
<td>Specifies the implementation of the message-passing interface (MPI) to use:</td>
</tr>
<tr>
<td>custom_library_name</td>
<td>const char *</td>
<td>The filename (without a directory name) of the custom BLACS dynamic library to use. This library must be located in the directory with your application executable or with Intel® MKL dynamic libraries. Can be NULL or an empty string.</td>
</tr>
</tbody>
</table>

Description

Call this function to set the MPI implementation to be used by Intel® MKL on Windows* OS when dynamic Intel® MKL libraries are used. For all other configurations, the function returns an error indicating that you cannot set the MPI implementation. You can specify your own prebuilt dynamic BLACS library for a custom MPI by setting `vendor` to `MKL_BLACS_CUSTOM` and optionally passing the name of the custom BLACS dynamic library. If the `custom_library_path` parameter is NULL or an empty string, Intel® MKL uses the default platform-specific library name: `mkl_blacs_custom_lp64.dll` or `mkl_blacs_custom_ilp64.dll`, depending on whether the BLACS interface linked against your application is LP64 or ILP64.

Return Values

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>The return status:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 - The function completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• -1 - The <code>vendor</code> parameter is invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• -2 - The <code>custom_library_name</code> parameter is invalid.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• -3 - The MPI library cannot be set at this point.</td>
</tr>
</tbody>
</table>

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain
### Optimization Notice

Optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### mkl_finalize

*Terminates Intel® MKL execution environment and frees resources allocated by the library.*

#### Syntax

```c
void mkl_finalize(void);
```

#### Include Files

- `mkl.h`

#### Description

This function frees resources allocated by Intel® MKL. Once this function is called, the application can no longer call Intel® MKL functions other than `mkl_finalize`.

In particular, the `mkl_finalize` function enables you to free resources when a third-party shared library is statically linked to Intel® MKL. To avoid resource leaks that may happen when a shared library is loaded and unloaded multiple times, call `mkl_finalize` each time the library is unloaded. The recommended method to do this depends on the operating system:

- On Linux* or macOS*, place the call into a shared library destructor.
- On Windows*, call `mkl_finalize` from the `DLL_PROCESS_DETACH` handler of `DllMain`.

#### NOTE

Intel® MKL shared libraries automatically perform finalization when they are unloaded. If an application is statically linked to Intel® MKL, the operating system frees all resources allocated by Intel® MKL during termination of the process associated with the application.
**BLACS Routines**

This chapter describes the Intel® Math Kernel Library implementation of FORTRAN 77 routines from the BLACS (Basic Linear Algebra Communication Subprograms) package. These routines are used to support a linear algebra oriented message passing interface that may be implemented efficiently and uniformly across a large range of distributed memory platforms.

The BLACS routines make linear algebra applications both easier to program and more portable. For this purpose, they are used in Intel® MKL intended for the Linux* and Windows* OSs as the communication layer of ScalAPACK and Cluster FFT.

On computers, a linear algebra matrix is represented by a two dimensional array (2D array), and therefore the BLACS operate on 2D arrays. See description of the basic matrix shapes in a special section.

The BLACS routines implemented in Intel® MKL are of four categories:

- Combines
- Point to Point Communication
- Broadcast
- Support.

The Combines take data distributed over processes and combine the data to produce a result. The Point to Point routines are intended for point-to-point communication and Broadcast routines send data possessed by one process to all processes within a scope.

The Support routines perform distinct tasks that can be used for initialization, destruction, information, and miscellaneous tasks.

### Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### Matrix Shapes

The BLACS routines recognize the two most common classes of matrices for dense linear algebra. The first of these classes consists of general rectangular matrices, which in machine storage are 2D arrays consisting of $m$ rows and $n$ columns, with a leading dimension, $lda$, that determines the distance between successive columns in memory.

The general rectangular matrices take the following parameters as input when determining what array to operate on:

- $m$ (input) INTEGER. The number of matrix rows to be operated on.
- $n$ (input) INTEGER. The number of matrix columns to be operated on.
- $a$ (input/output) TYPE (depends on routine), array of dimension ($lda, n$). A pointer to the beginning of the (sub)array to be sent.
lda (input) INTEGER. The distance between two elements in matrix row.

The second class of matrices recognized by the BLACS are trapezoidal matrices (triangular matrices are a sub-class of trapezoidal). Trapezoidal arrays are defined by $m$, $n$, and $lda$, as above, but they have two additional parameters as well. These parameters are:

uplo (input) CHARACTER*1. Indicates whether the matrix is upper or lower trapezoidal, as discussed below.

diag (input) CHARACTER*1. Indicates whether the diagonal of the matrix is unit diagonal (will not be operated on) or otherwise (will be operated on).

The shape of the trapezoidal arrays is determined by these parameters as follows:

__border__top

Trapezoidal Arrays Shapes

<table>
<thead>
<tr>
<th>uplo</th>
<th>$m \leq n$</th>
<th>$m &gt; n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>“U”</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td></td>
<td>$m$</td>
<td>$m$</td>
</tr>
<tr>
<td></td>
<td>$n - m$</td>
<td>$m - n$</td>
</tr>
</tbody>
</table>

The packing of arrays, if required, so that they may be sent efficiently is hidden, allowing the user to concentrate on the logical matrix, rather than on how the data is organized in the system memory.

Repeatability and Coherence

Floating point computations are not exact on almost all modern architectures. This lack of precision is particularly problematic in parallel operations. Since floating point computations are inexact, algorithms are classified according to whether they are repeatable and to what degree they guarantee coherence.

- Repeatable: a routine is repeatable if it is guaranteed to give the same answer if called multiple times with the same parallel configuration and input.
- Coherent: a routine is coherent if all processes selected to receive the answer get identical results.
NOTE
Repeatability and coherence do not effect correctness. A routine may be both incoherent and non-repeatable, and still give correct output. But inaccuracies in floating point calculations may cause the routine to return differing values, all of which are equally valid.

Repeatability
Because the precision of floating point arithmetic is limited, it is not truly associative: \((a + b) + c\) might not be the same as \(a + (b + c)\). The lack of exact arithmetic can cause problems whenever the possibility for reordering of floating point calculations exists. This problem becomes prevalent in parallel computing due to race conditions in message passing. For example, consider a routine which sums numbers stored on different processes. Assume this routine runs on four processes, with the numbers to be added being the process numbers themselves. Therefore, process 0 has the value 0:0, process 1 has the value 1:0, and so on.

One algorithm for the computation of this result is to have all processes send their process numbers to process 0; process 0 adds them up, and sends the result back to all processes. So, process 0 would add a number to 0:0 in the first step. If receiving the process numbers is ordered so that process 0 always receives the message from process 1 first, then 2, and finally 3, this results in a repeatable algorithm, which evaluates the expression \(((0:0 + 1:0) + 2:0) + 3:0\).

However, to get the best parallel performance, it is better not to require a particular ordering, and just have process 0 add the first available number to its value and continue to do so until all numbers have been added in. Using this method, a race condition occurs, because the order of the operation is determined by the order in which process 0 receives the messages, which can be effected by any number of things. This implementation is not repeatable, because the answer can vary between invocations, even if the input is the same. For instance, one run might produce the sequence \(((0:0 + 1:0) + 2:0) + 3:0\), while a subsequent run could produce \(((0:0 + 2:0 + 1:0) + 3:0\). Both of these results are correct summations of the given numbers, but because of floating point roundoff, they might be different.

Coherence
A routine produces coherent output if all processes are guaranteed to produce the exact same results. Obviously, almost no algorithm involving communication is coherent if communication can change the values being communicated. Therefore, if the parallel system being studied cannot guarantee that communication between processes preserves values, no routine is guaranteed to produce coherent results.

If communication is assumed to be coherent, there are still various levels of coherent algorithms. Some algorithms guarantee coherence only if floating point operations are done in the exact same order on every node. This is homogeneous coherence: the result will be coherent if the parallel machine is homogeneous in its handling of floating point operations.

A stronger assertion of coherence is heterogeneous coherence, which does not require all processes to have the same handling of floating point operations.

In general, a routine that is homogeneous coherent performs computations redundantly on all nodes, so that all processes get the same answer only if all processes perform arithmetic in the exact same way, whereas a routine which is heterogeneous coherent is usually constrained to having one process calculate the final result, and broadcast it to all other processes.

Example of Incoherence
An incoherent algorithm is one which does not guarantee that all processes get the same result even on a homogeneous system with coherent communication. The previous example of summing the process numbers demonstrates this kind of behavior. One way to perform such a sum is to have every process broadcast its number to all other processes. Each process then adds these numbers, starting with its own. The calculations performed by each process receives would then be:
• Process 0: \(((0:0+ 1:0) + 2:0) + 3:0\)
• Process 1: \(((1:0+ 2:0) + 3:0) + 0:0\)
• Process 2: \(((2:0+ 3:0) + 0:0) + 1:0\)
• Process 3: \(((3:0+ 0:0) + 1:0) + 0:0\)

All of these results are equally valid, and since all the results might be different from each other, this algorithm is incoherent. Notice, however, that this algorithm is repeatable: each process will get the same result if the algorithm is called again on the same data.

Example of Homogeneous Coherence
Another way to perform this summation is for all processes to send their data to all other processes, and to ensure the result is not incoherent, enforce the ordering so that the calculation each node performs is \(((0:0+ 1:0) + 2:0) + 3:0\). This answer is the same for all processes only if all processes do the floating point arithmetic in the same way. Otherwise, each process may make different floating point errors during the addition, leading to incoherence of the output. Notice that since there is a specific ordering to the addition, this algorithm is repeatable.

Example of Heterogeneous Coherence
In the final example, all processes send the result to process 0, which adds the numbers and broadcasts the result to the rest of the processes. Since one process does all the computation, it can perform the operations in any order and it will give coherent results as long as communication is itself coherent. If a particular order is not forced on the the addition, the algorithm will not be repeatable. If a particular order is forced, it will be repeatable.

Summary
Repeatability and coherence are separate issues which may occur in parallel computations. These concepts may be summarized as:

• Repeatability: The routine will yield the exact same result if it run multiple times on an identical problem. Each process may get a different result than the others (i.e., repeatability does not imply coherence), but that value will not change if the routine is invoked multiple times.
• Homogeneous coherence: All processes selected to possess the result will receive the exact same answer if:
  • Communication does not change the value of the communicated data.
  • All processes perform floating point arithmetic exactly the same.
• Heterogeneous coherence: All processes will receive the exact same answer if communication does not change the value of the communicated data.

In general, lack of the associative property for floating point calculations may cause both incoherence and non-repeatability. Algorithms that rely on redundant computations are at best homogeneous coherent, and algorithms in which one process broadcasts the result are heterogeneous coherent. Repeatability does not imply coherence, nor does coherence imply repeatability.

Since these issues do not effect the correctness of the answer, they can usually be ignored. However, in very specific situations, these issues may become very important. A stopping criteria should not be based on incoherent results, for instance. Also, a user creating and debugging a parallel program may wish to enforce repeatability so the exact same program sequence occurs on every run.

In the BLACS, coherence and repeatability apply only in the context of the combine operations. As mentioned above, it is possible to have communication which is incoherent (for instance, two machines which store floating point numbers differently may easily produce incoherent communication, since a number stored on machine A may not have a representation on machine B). However, the BLACS cannot control this issue. Communication is assumed to be coherent, which for communication implies that it is also repeatable.
For combine operations, the BLACS allow you to set flags indicating that you would like combines to be repeatable and/or heterogeneous coherent (see `blacs_get` and `blacs_set` for details on setting these flags).

If the BLACS are instructed to guarantee heterogeneous coherency, the BLACS restrict the topologies which can be used so that one process calculates the final result of the combine, and if necessary, broadcasts the answer to all other processes.

If the BLACS are instructed to guarantee repeatability, orderings will be enforced in the topologies which are selected. This may result in loss of performance which can range from negligible to serious depending on the application.

A couple of additional notes are in order. Incoherence and nonrepeatability can arise as a result of floating point errors, as discussed previously. This might lead you to suspect that integer calculations are always repeatable and coherent, since they involve exact arithmetic. This is true if overflow is ignored. With overflow taken into consideration, even integer calculations can display incoherence and non-repeatability. Therefore, if the repeatability or coherence flags are set, the BLACS treats integer combines the same as floating point combines in enforcing repeatability and coherence guards.

By their nature, maximization and minimization should always be repeatable. In the complex precisions, however, the real and imaginary parts must be combined in order to obtain a magnitude value used to do the comparison (this is typically $|r| + |i|$ or $\sqrt{r^2 + i^2}$). This allows for the possibility of heterogeneous incoherence. The BLACS therefore restrict which topologies are used for maximization and minimization in the complex routines when the heterogeneous coherence flag is set.

**BLACS Combine Operations**

This section describes BLACS routines that combine the data to produce a result.

In a combine operation, each participating process contributes data that is combined with other processes’ data to produce a result. This result can be given to a particular process (called the *destination* process), or to all participating processes. If the result is given to only one process, the operation is referred to as a *leave-on-one* combine, and if the result is given to all participating processes the operation is referenced as a *leave-on-all* combine.

At present, three kinds of combines are supported. They are:

- element-wise summation
- element-wise absolute value maximization
- element-wise absolute value minimization

of general rectangular arrays.

Note that a combine operation combines data between processes. By definition, a combine performed across a scope of only one process does not change the input data. This is why the operations (max/min/sum) are specified as *element-wise*. Element-wise indicates that each element of the input array will be combined with the corresponding element from all other processes’ arrays to produce the result. Thus, a 4 x 2 array of inputs produces a 4 x 2 answer array.

When the max/min comparison is being performed, absolute value is used. For example, -5 and 5 are equivalent. However, the returned value is unchanged; that is, it is not the absolute value, but is a signed value instead. Therefore, if you performed a BLACS absolute value maximum combine on the numbers -5, 3, 1, 8 the result would be -8.

The initial symbol ? in the routine names below masks the data type:

- `i` integer
- `s` single precision real
- `d` double precision real
- `c` single precision complex
BLACS Combines

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Results of operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>gamx2d</td>
<td>Entries of result matrix will have the value of the greatest absolute value found in that position.</td>
</tr>
<tr>
<td>gamn2d</td>
<td>Entries of result matrix will have the value of the smallest absolute value found in that position.</td>
</tr>
<tr>
<td>gsum2d</td>
<td>Entries of result matrix will have the summation of that position.</td>
</tr>
</tbody>
</table>

?gamx2d

Performs element-wise absolute value maximization.

Syntax

call igamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call sgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call dgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call cgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
call zgamx2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )

Input Parameters

icontxt       INTEGER. Integer handle that indicates the context.
scope         CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.
top           CHARACTER*1. Communication pattern to use during the combine operation.
m             INTEGER. The number of matrix rows to be combined.
n             INTEGER. The number of matrix columns to be combined.
a             TYPE array (lda, n). Matrix to be compared with to produce the maximum.
lda           INTEGER. The leading dimension of the matrix A, that is, the distance between two successive elements in a matrix row.
rcflag        INTEGER.

If rcflag = -1, the arrays ra and ca are not referenced and need not exist. Otherwise, rcflag indicates the leading dimension of these arrays, and so must be ≥ m.
rdest         INTEGER.

The process row coordinate of the process that should receive the result. If rdest or cdest = -1, all processes within the indicated scope receive the answer.
The process column coordinate of the process that should receive the result. If \( rdest \) or \( cdest \) = -1, all processes within the indicated scope receive the answer.

**Output Parameters**

\( a \)  
**TYPE** array \((lda, n)\). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

\( ra \)  
**INTEGER** array \((rcflag, n)\). If \( rcflag = -1 \), this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least \( rcflag \times n \)) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

\( ca \)  
**INTEGER** array \((rcflag, n)\). If \( rcflag = -1 \), this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least \( rcflag \times n \)) indicating the row index of the process that provided the maximum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

**Description**

This routine performs element-wise absolute value maximization, that is, each element of matrix \( A \) is compared with the corresponding element of the other process's matrices. Note that the value of \( A \) is returned, but the absolute value is used to determine the maximum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

**See Also**

Examples of BLACS Routines Usage

\(?\text{gamn2d}\)  
*Performs element-wise absolute value minimization.*

**Syntax**

```c
  call igamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
  call sgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
  call dgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
  call cgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
  call zgamn2d( icontxt, scope, top, m, n, a, lda, ra, ca, rcflag, rdest, cdest )
```

**Input Parameters**

\( icontxt \)  
**INTEGER.** Integer handle that indicates the context.
**Scope**

**top**

**m**

**n**

**a**

**lda**

**rcflag**

**rdest**

**cdest**

**Output Parameters**

**a**

**ra**

**ca**

**scope**

CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.

**top**

CHARACTER*1. Communication pattern to use during the combine operation.

**m**

INTEGER. The number of matrix rows to be combined.

**n**

INTEGER. The number of matrix columns to be combined.

**a**

TYPE array (lda, n). Matrix to be compared with to produce the minimum.

**lda**

INTEGER. The leading dimension of the matrix A, that is, the distance between two successive elements in a matrix row.

**rcflag**

If rcflag = -1, the arrays ra and ca are not referenced and need not exist. Otherwise, rcflag indicates the leading dimension of these arrays, and so must be ≥ m.

**rdest**

INTEGER. The process row coordinate of the process that should receive the result. If rdest or cdest = -1, all processes within the indicated scope receive the answer.

**cdest**

INTEGER. The process column coordinate of the process that should receive the result. If rdest or cdest = -1, all processes within the indicated scope receive the answer.

**Output Parameters**

**a**

TYPE array (lda, n). Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

**ra**

INTEGER array (rcflag, n).

If rcflag = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least rcflag x n) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.

**ca**

INTEGER array (rcflag, n).

If rcflag = -1, this array will not be referenced, and need not exist. Otherwise, it is an integer array (of size at least rcflag x n) indicating the row index of the process that provided the minimum. If the calling process is not selected to receive the result, this array will contain intermediate (useless) results.
Description
This routine performs element-wise absolute value minimization, that is, each element of matrix \( A \) is compared with the corresponding element of the other process's matrices. Note that the value of \( A \) is returned, but the absolute value is used to determine the minimum (the 1-norm is used for complex numbers). Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also
Examples of BLACS Routines Usage

?gsum2d
Performs element-wise summation.

Syntax
```
call igsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call sgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call dgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call cgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
call zgsum2d( icontxt, scope, top, m, n, a, lda, rdest, cdest )
```

Input Parameters

icontxt
INTEGER. Integer handle that indicates the context.

scope
CHARACTER*1. Indicates what scope the combine should proceed on. Limited to ROW, COLUMN, or ALL.

top
CHARACTER*1. Communication pattern to use during the combine operation.

m
INTEGER. The number of matrix rows to be combined.

n
INTEGER. The number of matrix columns to be combined.

a
TYPE array (lda, n). Matrix to be added to produce the sum.

lda
INTEGER. The leading dimension of the matrix \( A \), that is, the distance between two successive elements in a matrix row.

rdest
INTEGER. The process row coordinate of the process that should receive the result. If \( rdest \) or \( cdest = -1 \), all processes within the indicated scope receive the answer.

cdest
INTEGER. The process column coordinate of the process that should receive the result. If \( rdest \) or \( cdest = -1 \), all processes within the indicated scope receive the answer.
Output Parameters

\[ a \quad \text{TYPE array (lda, n).} \]

Contains the result if this process is selected to receive the answer, or intermediate results if the process is not selected to receive the result.

Description

This routine performs element-wise summation, that is, each element of matrix \( A \) is summed with the corresponding element of the other process's matrices. Combines may be globally-blocking, so they must be programmed as if no process returns until all have called the routine.

See Also

Examples of BLACS Routines Usage

BLACS Point To Point Communication

This section describes BLACS routines for point to point communication.

Point to point communication requires two complementary operations. The \textit{send} operation produces a message that is then consumed by the \textit{receive} operation. These operations have various resources associated with them. The main such resource is the buffer that holds the data to be sent or serves as the area where the incoming data is to be received. The level of \textit{blocking} indicates what correlation the return from a send/receive operation has with the availability of these resources and with the status of message.

Non-blocking

The return from the \textit{send} or \textit{receive} operations does not imply that the resources may be reused, that the message has been sent/received or that the complementary operation has been called. Return means only that the send/receive has been started, and will be completed at some later date. Polling is required to determine when the operation has finished.

In non-blocking message passing, the concept of \textit{communication/computation overlap} (abbreviated C/C overlap) is important. If a system possesses C/C overlap, independent computation can occur at the same time as communication. That means a nonblocking operation can be posted, and unrelated work can be done while the message is sent/received in parallel. If C/C overlap is not present, after returning from the routine call, computation will be interrupted at some later date when the message is actually sent or received.

Locally-blocking

Return from the \textit{send} or \textit{receive} operations indicates that the resources may be reused. However, since this only depends on local information, it is unknown whether the complementary operation has been called. There are no locally-blocking receives: the send must be completed before the receive buffer is available for re-use.

If a receive has not been posted at the time a locally-blocking send is issued, buffering will be required to avoid losing the message. Buffering can be done on the sending process, the receiving process, or not done at all, losing the message.

Globally-blocking

Return from a globally-blocking procedure indicates that the operation resources may be reused, and that complement of the operation has at least been posted. Since the receive has been posted, there is no buffering required for globally-blocking sends: the message is always sent directly into the user's receive buffer.
Almost all processors support non-blocking communication, as well as some other level of blocking sends. What level of blocking the send possesses varies between platforms. For instance, the Intel® processors support locally-blocking sends, with buffering done on the receiving process. This is a very important distinction, because codes written assuming locally-blocking sends will hang on platforms with globally-blocking sends. Below is a simple example of how this can occur:

```plaintext
IAM = MY_PROCESS_ID()
IF (IAM .EQ. 0) THEN
    SEND TO PROCESS 1
    RECV FROM PROCESS 1
ELSE IF (IAM .EQ. 1) THEN
    SEND TO PROCESS 0
    RECV FROM PROCESS 0
END IF
```

If the send is globally-blocking, process 0 enters the send, and waits for process 1 to start its receive before continuing. In the meantime, process 1 starts to send to 0, and waits for 0 to receive before continuing. Both processes are now waiting on each other, and the program will never continue.

The solution for this case is obvious. One of the processes simply reverses the order of its communication calls and the hang is avoided. However, when the communication is not just between two processes, but rather involves a hierarchy of processes, determining how to avoid this kind of difficulty can become problematic.

For this reason, it was decided the BLACS would support locally-blocking sends. On systems natively supporting globally-blocking sends, non-blocking sends coupled with buffering is used to simulate locally-blocking sends. The BLACS support globally-blocking receives.

In addition, the BLACS specify that point to point messages between two given processes will be strictly ordered. If process 0 sends three messages (label them A, B, and C) to process 1, process 1 must receive A before it can receive B, and message C can be received only after both A and B. The main reason for this restriction is that it allows for the computation of message identifiers.

Note, however, that messages from different processes are not ordered. If processes 0, . . ., 3 send messages A, . . ., D to process 4, process 4 may receive these messages in any order that is convenient.

**Convention**

The convention used in the communication routine names follows the template ?xxyy2d, where the letter in the ? position indicates the data type being sent, xx is replaced to indicate the shape of the matrix, and the yy positions are used to indicate the type of communication to perform:

- i: integer
- s: single precision real
- d: double precision real
- c: single precision complex
- z: double precision complex
- ge: The data to be communicated is stored in a general rectangular matrix.
- tr: The data to be communicated is stored in a trapezoidal matrix.
- sd: Send. One process sends to another.
- rv: Receive. One process receives from another.
## BLACS Point To Point Communication

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gesd2d</td>
<td>Take the indicated matrix and send it to the destination process.</td>
</tr>
<tr>
<td>trsd2d</td>
<td>Take the indicated matrix and send it to the destination process.</td>
</tr>
<tr>
<td>gerv2d</td>
<td>Receive a message from the process into the matrix.</td>
</tr>
<tr>
<td>trrv2d</td>
<td>Receive a message from the process into the matrix.</td>
</tr>
</tbody>
</table>

As a simple example, the pseudo code given above is rewritten below in terms of the BLACS. It is further specified that the data being exchanged is the double precision vector \( X \), which is 5 elements long.

```fortran
CALL GRIDINFO(NPROW, NPCOL, MYPROW, MYPCOL)
IF (MYPROW.EQ.0 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 1, 0)
    CALL DGERV2D(5, 1, X, 5, 1, 0)
ELSE IF (MYPROW.EQ.1 .AND. MYPCOL.EQ.0) THEN
    CALL DGESD2D(5, 1, X, 5, 0, 0)
    CALL DGERV2D(5, 1, X, 5, 0, 0)
END IF
```

### gesd2d

*Take a general rectangular matrix and sends it to the destination process.*

#### Syntax

```fortran
call igesd2d( icontxt, m, n, a, lda, rdest, cdest )
call sgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call dgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call cgesd2d( icontxt, m, n, a, lda, rdest, cdest )
call zgesd2d( icontxt, m, n, a, lda, rdest, cdest )
```

#### Input Parameters

- `icontxt` _INTEGER_. Integer handle that indicates the context.
- `m, n, a, lda` Describe the matrix to be sent. See Matrix Shapes for details.
- `rdest` _INTEGER_. The process row coordinate of the process to send the message to.
- `cdest` _INTEGER_. The process column coordinate of the process to send the message to.

#### Description

This routine takes the indicated general rectangular matrix and sends it to the destination process located at \( \{RDEST, CDEST\} \) in the process grid. Return from the routine indicates that the buffer (the matrix \( A \)) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

#### See Also

Examples of BLACS Routines Usage
?trsd2d
Takes a trapezoidal matrix and sends it to the
destination process.

Syntax

```plaintext
call itrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call strsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call dtrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ctrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
call ztrsd2d( icontxt, uplo, diag, m, n, a, lda, rdest, cdest )
```

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>icontxt</td>
<td>INTEGER. Integer handle that indicates the context.</td>
</tr>
<tr>
<td>uplo, diag, m, n, a, lda</td>
<td>Describe the matrix to be sent. See Matrix Shapes for details.</td>
</tr>
<tr>
<td>rdest</td>
<td>INTEGER. The process row coordinate of the process to send the message to.</td>
</tr>
<tr>
<td>cdest</td>
<td>INTEGER. The process column coordinate of the process to send the message to.</td>
</tr>
</tbody>
</table>

Description

This routine takes the indicated trapezoidal matrix and sends it to the destination process located at \{RDEST, CDEST\} in the process grid. Return from the routine indicates that the buffer (the matrix A) may be reused. The routine is locally-blocking, that is, it will return even if the corresponding receive is not posted.

?gerv2d
Receives a message from the process into the general rectangular matrix.

Syntax

```plaintext
call igerv2d( icontxt, m, n, a, lda, rsr, csr )
call sgerv2d( icontxt, m, n, a, lda, rsr, csr )
call dgerv2d( icontxt, m, n, a, lda, rsr, csr )
call cgerv2d( icontxt, m, n, a, lda, rsr, csr )
call zgerv2d( icontxt, m, n, a, lda, rsr, csr )
```

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>icontxt</td>
<td>INTEGER. Integer handle that indicates the context.</td>
</tr>
<tr>
<td>m, n, lda</td>
<td>INTEGER.</td>
</tr>
<tr>
<td>rsr</td>
<td>INTEGER. Integer handle that indicates the context.</td>
</tr>
</tbody>
</table>

Describe the matrix to be sent. See Matrix Shapes for details.
The process row coordinate of the source of the message.

\[ csr \]

The process column coordinate of the source of the message.

**Output Parameters**

\[ a \]

An array of dimension \((lda, n)\) to receive the incoming message into.

**Description**

This routine receives a message from process \{RSRC, CSRC\} into the general rectangular matrix \(A\). This routine is globally-blocking, that is, return from the routine indicates that the message has been received into \(A\).

**See Also**

Examples of BLACS Routines Usage

**?trrv2d**

*Receives a message from the process into the trapezoidal matrix.*

**Syntax**

\[
\text{call itrv2d}( \text{icontxt}, \text{uplo}, \text{diag}, m, n, a, lda, rsr, csr ) \\
\text{call strv2d}( \text{icontxt}, \text{uplo}, \text{diag}, m, n, a, lda, rsr, csr ) \\
\text{call dtrv2d}( \text{icontxt}, \text{uplo}, \text{diag}, m, n, a, lda, rsr, csr ) \\
\text{call ctrv2d}( \text{icontxt}, \text{uplo}, \text{diag}, m, n, a, lda, rsr, csr ) \\
\text{call ztrv2d}( \text{icontxt}, \text{uplo}, \text{diag}, m, n, a, lda, rsr, csr )
\]

**Input Parameters**

\[ \text{icontxt} \]

INTEGER. Integer handle that indicates the context.

\[ \text{uplo}, \text{diag}, m, n, lda \]

Describe the matrix to be sent. See Matrix Shapes for details.

\[ \text{rsr} \]

INTEGER.

The process row coordinate of the source of the message.

\[ \text{csr} \]

INTEGER.

The process column coordinate of the source of the message.

**Output Parameters**

\[ a \]

An array of dimension \((lda, n)\) to receive the incoming message into.

**Description**

This routine receives a message from process \{RSRC, CSRC\} into the trapezoidal matrix \(A\). This routine is globally-blocking, that is, return from the routine indicates that the message has been received into \(A\).
BLACS Broadcast Routines

This section describes BLACS broadcast routines.

A broadcast sends data possessed by one process to all processes within a scope. Broadcast, much like point to point communication, has two complementary operations. The process that owns the data to be broadcast issues a broadcast/send. All processes within the same scope must then issue the complementary broadcast/receive.

The BLACS define that both broadcast/send and broadcast/receive are globally-blocking. Broadcasts/receives cannot be locally-blocking since they must post a receive. Note that receives cannot be locally-blocking. When a given process can leave, a broadcast/receive operation is topology dependent, so, to avoid a hang as topology is varied, the broadcast/receive must be treated as if no process can leave until all processes have called the operation.

Broadcast/sends could be defined to be locally-blocking. Since no information is being received, as long as locally-blocking point to point sends are used, the broadcast/send will be locally blocking. However, defining one process within a scope to be locally-blocking while all other processes are globally-blocking adds little to the programmability of the code. On the other hand, leaving the option open to have globally-blocking broadcast/sends may allow for optimization on some platforms.

The fact that broadcasts are defined as globally-blocking has several important implications. The first is that scoped operations (broadcasts or combines) must be strictly ordered, that is, all processes within a scope must agree on the order of calls to separate scoped operations. This constraint falls in line with that already in place for the computation of message IDs, and is present in point to point communication as well.

A less obvious result is that scoped operations with \texttt{SCOPE} = 'ALL' must be ordered with respect to any other scoped operation. This means that if there are two broadcasts to be done, one along a column, and one involving the entire process grid, all processes within the process column issuing the column broadcast must agree on which broadcast will be performed first.

The convention used in the communication routine names follows the template \texttt{?xxyy2d}, where the letter in the \texttt{?} position indicates the data type being sent, \texttt{xx} is replaced to indicate the shape of the matrix, and the \texttt{yy} positions are used to indicate the type of communication to perform:

<table>
<thead>
<tr>
<th>\texttt{i}</th>
<th>\texttt{s}</th>
<th>\texttt{d}</th>
<th>\texttt{c}</th>
<th>\texttt{z}</th>
<th>\texttt{ge}</th>
<th>\texttt{tr}</th>
<th>\texttt{bs}</th>
<th>\texttt{br}</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>single precision real</td>
<td>double precision real</td>
<td>single precision complex</td>
<td>double precision complex</td>
<td>The data to be communicated is stored in a general rectangular matrix.</td>
<td>The data to be communicated is stored in a trapezoidal matrix.</td>
<td>Broadcast/send. A process begins the broadcast of data within a scope.</td>
<td>Broadcast/receive A process receives and participates in the broadcast of data within a scope.</td>
</tr>
</tbody>
</table>

### BLACS Broadcast Routines

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>gebs2d</td>
<td>Start a broadcast along a scope.</td>
</tr>
<tr>
<td>trbs2d</td>
<td></td>
</tr>
<tr>
<td>gebr2d</td>
<td>Receive and participate in a broadcast along a scope.</td>
</tr>
<tr>
<td>trbr2d</td>
<td></td>
</tr>
</tbody>
</table>
Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

?gebs2d

Starts a broadcast along a scope for a general rectangular matrix.

Syntax

\[
\begin{align*}
\text{call } & \text{igebs2d}( \text{icontxt}, \text{scope}, \text{top}, m, n, a, \text{lda} ) \\
\text{call } & \text{sgebs2d}( \text{icontxt}, \text{scope}, \text{top}, m, n, a, \text{lda} ) \\
\text{call } & \text{dgebs2d}( \text{icontxt}, \text{scope}, \text{top}, m, n, a, \text{lda} ) \\
\text{call } & \text{cgebs2d}( \text{icontxt}, \text{scope}, \text{top}, m, n, a, \text{lda} ) \\
\text{call } & \text{zgebs2d}( \text{icontxt}, \text{scope}, \text{top}, m, n, a, \text{lda} )
\end{align*}
\]

Input Parameters

- icontxt: INTEGER. Integer handle that indicates the context.
- scope: CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
- top: CHARACTER*1. Indicates the communication pattern to use for the broadcast.
- m, n, a, lda

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the general rectangular matrix A.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

See Also

Examples of BLACS Routines Usage

?trbs2d

Starts a broadcast along a scope for a trapezoidal matrix.

Syntax

\[
\text{call itrbs2d}( \text{icontxt}, \text{scope}, \text{top}, \text{uplo}, \text{diag}, m, n, a, \text{lda} )
\]
call strbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call dtrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ctrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )
call ztrbs2d( icontxt, scope, top, uplo, diag, m, n, a, lda )

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

scope CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.

top CHARACTER*1. Indicates the communication pattern to use for the broadcast.

uplo, diag, m, 
n, a, lda

Description

This routine starts a broadcast along a scope. All other processes within the scope must call broadcast/receive for the broadcast to proceed. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix A.

Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

?gebr2d

Receives and participates in a broadcast along a scope for a general rectangular matrix.

Syntax

call igebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call sgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call dgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call cgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )
call zgebr2d( icontxt, scope, top, m, n, a, lda, rsrc, csrc )

Input Parameters

icontxt INTEGER. Integer handle that indicates the context.

scope CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.

top CHARACTER*1. Indicates the communication pattern to use for the broadcast.

m, n, lda Describe the matrix to be sent. See Matrix Shapes for details.

rsrc INTEGER. The process row coordinate of the process that called broadcast/send.
**Input Parameters**

- `icontxt` INTEGER. Integer handle that indicates the context.
- `scope` CHARACTER*1. Indicates what scope the broadcast should proceed on. Limited to 'Row', 'Column', or 'All'.
- `top` CHARACTER*1. Indicates the communication pattern to use for the broadcast.
- `uplo, diag, m, n, lda` Describe the matrix to be sent. See Matrix Shapes for details.
- `rsrc` INTEGER. The process row coordinate of the process that called broadcast/send.
- `csrc` INTEGER. The process column coordinate of the process that called broadcast/send.

**Output Parameters**

- `a` An array of dimension `(lda, n)` to receive the incoming message into.
Description
This routine receives and participates in a broadcast along a scope. At the end of a broadcast, all processes within the scope will possess the data in the trapezoidal matrix $A$. Broadcasts may be globally-blocking. This means no process is guaranteed to return from a broadcast until all processes in the scope have called the appropriate routine (broadcast/send or broadcast/receive).

BLACS Support Routines
The support routines perform distinct tasks that can be used for:
Initialization
Destruction
Information Purposes
Miscellaneous Tasks.

Initialization Routines
This section describes BLACS routines that deal with grid/context creation, and processing before the grid/context has been defined.

BLACS Initialization Routines

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>blacs_pinfo</td>
<td>Returns the number of processes available for use.</td>
</tr>
<tr>
<td>blacs_setup</td>
<td>Allocates virtual machine and spawns processes.</td>
</tr>
<tr>
<td>blacs_get</td>
<td>Gets values that BLACS use for internal defaults.</td>
</tr>
<tr>
<td>blacs_set</td>
<td>Sets values that BLACS use for internal defaults.</td>
</tr>
<tr>
<td>blacs_gridinit</td>
<td>Assigns available processes into BLACS process grid.</td>
</tr>
<tr>
<td>blacs_gridmap</td>
<td>Maps available processes into BLACS process grid.</td>
</tr>
</tbody>
</table>

blacs_pinfo

*Returns the number of processes available for use.*

Syntax

call blacs_pinfo( mynum, nprocs )

Output Parameters

mynum

INTEGER. An integer between 0 and $(nprocs - 1)$ that uniquely identifies each process.

nprocs

INTEGER. The number of processes available for BLACS use.

Description
This routine is used when some initial system information is required before the BLACS are set up. On all platforms except PVM, $nprocs$ is the actual number of processes available for use, that is, $nprows \times npcols \leq nprocs$. In PVM, the virtual machine may not have been set up before this call, and therefore no parallel machine exists. In this case, $nprocs$ is returned as less than one. If a process has been spawned via the
keyboard, it receives *mynum* of 0, and all other processes get *mynum* of -1. As a result, the user can distinguish between processes. Only after the virtual machine has been set up via a call to `BLACS_SETUP`, this routine returns the correct values for *mynum* and *nprocs*.

**See Also**

Examples of BLACS Routines Usage

**blacs_setup**

Allocates virtual machine and spawns processes.

**Syntax**

```plaintext
call blacs_setup( mynum, nprocs )
```

**Input Parameters**

*nprocs*  
*INTEGER.* On the process spawned from the keyboard rather than from `pvmspawn`, this parameter indicates the number of processes to create when building the virtual machine.

**Output Parameters**

*mynum*  
*INTEGER.* An integer between 0 and (*nprocs* - 1) that uniquely identifies each process.

*nprocs*  
*INTEGER.* For all processes other than spawned from the keyboard, this parameter means the number of processes available for BLACS use.

**Description**

This routine only accomplishes meaningful work in the PVM BLACS. On all other platforms, it is functionally equivalent to `blacs_pinfo`. The BLACS assume a static system, that is, the given number of processes does not change. PVM supplies a dynamic system, allowing processes to be added to the system on the fly.

`blacs_setup` is used to allocate the virtual machine and spawn off processes. It reads in a file called `blacs_setup.dat`, in which the first line must be the name of your executable. The second line is optional, but if it exists, it should be a PVM spawn flag. Legal values at this time are 0 (`PvmTaskDefault`), 4 (`PvmTaskDebug`), 8 (`PvmTaskTrace`), and 12 (`PvmTaskDebug + PvmTaskTrace`). The primary reason for this line is to allow the user to easily turn on and off PVM debugging. Additional lines, if any, specify what machines should be added to the current configuration before spawning *nprocs*-1 processes to the machines in a round robin fashion.

*nprocs* is input on the process which has no PVM parent (that is, *mynum*=0), and both parameters are output for all processes. So, on PVM systems, the call to `blacs_pinfo` informs you that the virtual machine has not been set up, and a call to `blacs_setup` then sets up the machine and returns the real values for *mynum* and *nprocs*.

Note that if the file `blacs_setup.dat` does not exist, the BLACS prompt the user for the executable name, and processes are spawned to the current PVM configuration.

**See Also**

Examples of BLACS Routines Usage

**blacs_get**

Gets values that BLACS use for internal defaults.
Syntax

```fortran
call blacs_get( icontxt, what, val )
```

Input Parameters

`icontxt`  
INTEGER. On values of `what` that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

`what`  
INTEGER. Indicates what BLACS internal(s) should be returned in `val`. Present options are:

- `what = 0`: Handle indicating default system context.
- `what = 1`: The BLACS message ID range.
- `what = 2`: The BLACS debug level the library was compiled with.
- `what = 10`: Handle indicating the system context used to define the BLACS context whose handle is `icontxt`.
- `what = 11`: Number of rings multiring broadcast topology is presently using.
- `what = 12`: Number of branches general tree broadcast topology is presently using.
- `what = 13`: Number of rings multiring combine topology is presently using.
- `what = 14`: Number of branches general tree combine topology is presently using.
- `what = 15`: Whether topologies are forced to be repeatable or not. A non-zero return value indicates that topologies are being forced to be repeatable. See Repeatability and Coherence for more information about repeatability.
- `what = 16`: Whether topologies are forced to be heterogenous coherent or not. A non-zero return value indicates that topologies are being forced to be heterogenous coherent. See Repeatability and Coherence for more information about coherence.

Output Parameters

`val`  
INTEGER. The value of the BLACS internal.

Description

This routine gets the values that the BLACS are using for internal defaults. Some values are tied to a BLACS context, and some are more general. The most common use is in retrieving a default system context for input into `blacs_gridinit` or `blacs_gridmap`.

Some systems, such as MPI*, supply their own version of context. For those users who mix system code with BLACS code, a BLACS context should be formed in reference to a system context. Thus, the grid creation routines take a system context as input. If you wish to have strictly portable code, you may use `blacs_get` to retrieve a default system context that will include all available processes. This value is not tied to a BLACS context, so the parameter `icontxt` is unused.

`blacs_get` returns information on three quantities that are tied to an individual BLACS context, which is passed in as `icontxt`. The information that may be retrieved is:

- The handle of the system context upon which this BLACS context was defined
- The number of rings for `TOP = 'M'` (multiring broadcast/combine)
• The number of branches for \texttt{TOP = 'T'} (general tree broadcast/general tree gather).
• Whether topologies are being forced to be repeatable or heterogenous coherent.

See Also
Examples of BLACS Routines Usage

**blacs_set**

*Sets values that BLACS use for internal defaults.*

**Syntax**

```
call blacs_set( icontxt, what, val )
```

**Input Parameters**

- \texttt{icontxt}  
  INTEGER. For values of \texttt{what} that are tied to a particular context, this parameter is the integer handle indicating the context. Otherwise, ignored.

- \texttt{what}  
  INTEGER. Indicates what BLACS internal(s) should be set. Present values are:
  • 1 = Set the BLACS message ID range
  • 11 = Number of rings for multiring broadcast topology to use
  • 12 = Number of branches for general tree broadcast topology to use
  • 13 = Number of rings for multiring combine topology to use
  • 14 = Number of branches for general tree combine topology to use
  • 15 = Force topologies to be repeatable or not
  • 16 = Force topologies to be heterogenous coherent or not

- \texttt{val}  
  INTEGER. Array of dimension (*). Indicates the value(s) the internals should be set to. The specific meanings depend on \texttt{what} values, as discussed in Description below.

**Description**

This routine sets the BLACS internal defaults depending on \texttt{what} values:

- **\texttt{what} = 1**

  Setting the BLACS message ID range.

  If you wish to mix the BLACS with other message-passing packages, restrict the BLACS to a certain message ID range not to be used by the non-BLACS routines. The message ID range must be set before the first call to \texttt{blacs_gridinit} or \texttt{blacs_gridmap}. Subsequent calls will have no effect. Because the message ID range is not tied to a particular context, the parameter \texttt{icontxt} is ignored, and \texttt{val} is defined as:

  \texttt{VAL (input) INTEGER array of dimension (2)}

  - \texttt{VAL(1)}: The smallest message ID (also called message type or message tag) the BLACS should use.
  - \texttt{VAL(2)}: The largest message ID (also called message type or message tag) the BLACS should use.

- **\texttt{what} = 11**

  Set number of rings for \texttt{TOP = 'M'} (multiring broadcast). This quantity is tied to a context, so \texttt{icontxt} is used, and \texttt{val} is defined as:
VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of rings for multiring topology to use.

**what = 12**

Set number of branches for TOP = 'T' (general tree broadcast). This quantity is tied to a context, so icontxt is used, and val is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of branches for general tree topology to use.

**what = 13**

Set number of rings for TOP = 'M' (multiring combine). This quantity is tied to a context, so icontxt is used, and val is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of rings for multiring topology to use.

**what = 14**

Set number of branches for TOP = 'T' (general tree gather). This quantity is tied to a context, so icontxt is used, and val is defined as:

VAL (input) INTEGER array of dimension (1)

VAL(1) : The number of branches for general tree topology to use.

**what = 15**

Force topologies to be repeatable or not (see Repeatability and Coherence for more information about repeatability).

VAL (input) INTEGER array of dimension (1)

VAL(1) = 0 (default) Topologies are not required to be repeatable.

VAL(1) ≠ 0 All used topologies are required to be repeatable, which might degrade performance.

**what = 16**

Force topologies to be heterogenous coherent or not (see Repeatability and Coherence for more information about coherence).

VAL (input) INTEGER array of dimension (1)

VAL(1) = 0 (default) Topologies are not required to be heterogenous coherent.

VAL(1) ≠ 0 All used topologies are required to be heterogenous coherent, which might degrade performance.

---

**blacs_gridinit**

Assigns available processes into BLACS process grid.

**Syntax**

call blacs_gridinit( icontxt, layout, nprow, npcol )

**Input Parameters**

**icontxt** INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call blacs_get to obtain a default system context.
**layout**

CHARACTER*1. Indicates how to map processes to BLACS grid. Options are:
- 'R': Use row-major natural ordering
- 'C': Use column-major natural ordering
- ELSE: Use row-major natural ordering

**nprow**

INTEGER. Indicates how many process rows the process grid should contain.

**ncol**

INTEGER. Indicates how many process columns the process grid should contain.

### Output Parameters

**icontxt**

INTEGER. Integer handle to the created BLACS context.

### Description

All BLACS codes must call this routine, or its sister routine `blacs_gridmap`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine creates a simple \( nprow \times ncol \) process grid. This process grid uses the first \( nprow \times ncol \) processes, and assigns them to the grid in a row- or column-major natural ordering. If these process-to-grid mappings are unacceptable, call `blacs_gridmap`.

### See Also

Examples of BLACS Routines Usage
- blacs_get
- blacs_gridmap
- blacs_setup

**blacs_gridmap**

Maps available processes into BLACS process grid.

### Syntax

```call blacs_gridmap( icontxt, usermap, ldumap, nprow, ncol )```
**Input Parameters**

- **icontxt**: INTEGER. Integer handle indicating the system context to be used in creating the BLACS context. Call `blacs_get` to obtain a default system context.

- **usermap**: INTEGER. Array, dimension `(ldumap, npcol)`, indicating the process-to-grid mapping.

- **ldumap**: INTEGER. Leading dimension of the 2D array `usermap`. `ldumap ≥ nprow`.

- **nprow**: INTEGER. Indicates how many process rows the process grid should contain.

- **npcol**: INTEGER. Indicates how many process columns the process grid should contain.

**Output Parameters**

- **icontxt**: INTEGER. Integer handle to the created BLACS context.

**Description**

All BLACS codes must call this routine, or its sister routine `blacs_gridinit`. These routines take the available processes, and assign, or map, them into a BLACS process grid. In other words, they establish how the BLACS coordinate system maps into the native machine process numbering system. Each BLACS grid is contained in a context, so that it does not interfere with distributed operations that occur within other grids/contexts. These grid creation routines may be called repeatedly to define additional contexts/grids.

The creation of a grid requires input from all processes that are defined to be in this grid. Processes belonging to more than one grid have to agree on which grid formation will be serviced first, much like the globally blocking sum or broadcast.

These grid creation routines set up various internals for the BLACS, and one of them must be called before any calls are made to the non-initialization BLACS.

Note that these routines map already existing processes to a grid: the processes are not created dynamically. On most parallel machines, the processes are actual processors (hardware), and they are "created" when you run your executable. When using the PVM BLACS, if the virtual machine has not been set up yet, the routine `blacs_setup` should be used to create the virtual machine.

This routine allows the user to map processes to the process grid in an arbitrary manner. `usermap(i,j)` holds the process number of the process to be placed in `{i, j}` of the process grid. On most distributed systems, this process number is a machine defined number between 0 ... `nprow-1`. For PVM, these node numbers are the PVM TIDS (Task IDs). The `blacs_gridmap` routine is intended for an experienced user. The `blacs_gridinit` routine is much simpler. `blacs_gridinit` simply performs a gridmap where the first `nprow * npcol` processes are mapped into the current grid in a row-major natural ordering. If you are an experienced user, `blacs_gridmap` allows you to take advantage of your system's actual layout. That is, you can map nodes that are physically connected to be neighbors in the BLACS grid, etc. The `blacs_gridmap` routine also opens the way for multigirding: you can separate your nodes into arbitrary grids, join them together at some later date, and then re-split them into new grids. `blacs_gridmap` also provides the ability to make arbitrary grids or subgrids (for example, a "nearest neighbor" grid), which can greatly facilitate operations among processes that do not fall on a row or column of the main process grid.

**See Also**

Examples of BLACS Routines Usage
Destruction Routines
This section describes BLACS routines that destroy grids, abort processes, and free resources.

**BLACS Destruction Routines**

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>blacs_freebuff</td>
<td>Frees BLACS buffer.</td>
</tr>
<tr>
<td>blacs_gridexit</td>
<td>Frees a BLACS context.</td>
</tr>
<tr>
<td>blacs_abort</td>
<td>Aborts all processes.</td>
</tr>
<tr>
<td>blacs_exit</td>
<td>Frees all BLACS contexts and releases all allocated memory.</td>
</tr>
</tbody>
</table>

**blacs_freebuff**

*Frees BLACS buffer.*

**Syntax**

call blacs_freebuff( icontxt, wait )

**Input Parameters**

- **icontxt**
  
  INTEGER. Integer handle that indicates the BLACS context.

- **wait**
  
  INTEGER. Parameter indicating whether to wait for non-blocking operations or not. If equals 0, the operations should not be waited for; free only unused buffers. Otherwise, wait in order to free all buffers.

**Description**

This routine releases the BLACS buffer.

The BLACS have at least one internal buffer that is used for packing messages. The number of internal buffers depends on what platform you are running the BLACS on. On systems where memory is tight, keeping this buffer or buffers may become expensive. Call freebuff to release the buffer. However, the next call of a communication routine that requires packing reallocates the buffer.

The *wait* parameter determines whether the BLACS should wait for any non-blocking operations to be completed or not. If *wait* = 0, the BLACS free any buffers that can be freed without waiting. If *wait* is not 0, the BLACS free all internal buffers, even if non-blocking operations must be completed first.

**blacs_gridexit**

*Frees a BLACS context.*

**Syntax**

call blacs_gridexit( icontxt )

**Input Parameters**

- **icontxt**
  
  INTEGER. Integer handle that indicates the BLACS context to be freed.
**Description**

This routine frees a BLACS context.

Release the resources when contexts are no longer needed. After freeing a context, the context no longer exists, and its handle may be re-used if new contexts are defined.

---

**blacs_abort**

*Aborts all processes.*

**Syntax**

```latex
call blacs_abort( icontxt, errornum )
```

**Input Parameters**

- `icontxt` : INTEGER. Integer handle that indicates the BLACS context to be aborted.
- `errornum` : INTEGER. User-defined integer error number.

**Description**

This routine aborts all the BLACS processes, not only those confined to a particular context.

Use `blacs_abort` to abort all the processes in case of a serious error. Note that both parameters are input, but the routine uses them only in printing out the error message. The context handle passed in is not required to be a valid context handle.

---

**blacs_exit**

*Frees all BLACS contexts and releases all allocated memory.*

**Syntax**

```latex
call blacs_exit( continue )
```

**Input Parameters**

- `continue` : INTEGER. Flag indicating whether message passing continues after the BLACS are done. If `continue` is non-zero, the user is assumed to continue using the machine after completing the BLACS. Otherwise, no message passing is assumed after calling this routine.

**Description**

This routine frees all BLACS contexts and releases all allocated memory.

This routine should be called when a process has finished all use of the BLACS. The `continue` parameter indicates whether the user will be using the underlying communication platform after the BLACS are finished. This information is most important for the PVM BLACS. If `continue` is set to 0, then `pvm_exit` is called; otherwise, it is not called. Setting `continue` not equal to 0 indicates that explicit PVM send/recvs will be called after the BLACS routines are used. Make sure your code calls `pvm_exit`. PVM users should either call `blacs_exit` or explicitly call `pvm_exit` to avoid PVM problems.

---

**See Also**

Examples of BLACS Routines Usage
Informational Routines
This section describes BLACS routines that return information involving the process grid.

BLACS Informational Routines

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>blacs_gridinfo</td>
<td>Returns information on the current grid.</td>
</tr>
<tr>
<td>blacs_pnum</td>
<td>Returns the system process number of the process in the process grid.</td>
</tr>
<tr>
<td>blacs_pcoord</td>
<td>Returns the row and column coordinates in the process grid.</td>
</tr>
</tbody>
</table>

blacs_gridinfo
Returns information on the current grid.

Syntax

```call blacs_gridinfo( icontxt, nprow, npcol, myprow, mypcol )```

Input Parameters

- `icontxt`  INTEGER. Integer handle that indicates the context.

Output Parameters

- `nprow`  INTEGER. Number of process rows in the current process grid.
- `npcol`  INTEGER. Number of process columns in the current process grid.
- `myprow`  INTEGER. Row coordinate of the calling process in the process grid.
- `mypcol`  INTEGER. Column coordinate of the calling process in the process grid.

Description

This routine returns information on the current grid. If the context handle does not point at a valid context, all quantities are returned as -1.

See Also

Examples of BLACS Routines Usage

blacs_pnum
Returns the system process number of the process in the process grid.

Syntax

```call blacs_pnum( icontxt, prow, pcol )```

Input Parameters

- `icontxt`  INTEGER. Integer handle that indicates the context.

- `prow`  INTEGER. Row coordinate of the process the system process number of which is to be determined.
Description
This function returns the system process number of the process at \{PROW, PCOL\} in the process grid.

See Also
Examples of BLACS Routines Usage

blacs_pcoord
Returns the row and column coordinates in the process grid.

Syntax

call blacs_pcoord( icontxt, pnum, prow, pcol )

Input Parameters

icontxt
INTEGER. Integer handle that indicates the context.

pnum
INTEGER. Process number the coordinates of which are to be determined. This parameter stand for the process number of the underlying machine, that is, it is a \texttt{tid} for PVM.

Output Parameters

prow
INTEGER. Row coordinates of the \texttt{pnum} process in the BLACS grid.

pcol
INTEGER. Column coordinates of the \texttt{pnum} process in the BLACS grid.

Description
Given the system process number, this function returns the row and column coordinates in the BLACS process grid.

See Also
Examples of BLACS Routines Usage

Miscellaneous Routines
This section describes \texttt{blacs_barrier} routine.

BLACS Informational Routines

<table>
<thead>
<tr>
<th>Routine name</th>
<th>Operation performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>blacs_barrier</td>
<td>Holds up execution of all processes within the indicated scope until they have all called the routine.</td>
</tr>
</tbody>
</table>

\texttt{blacs_barrier}
\textit{Holds up execution of all processes within the indicated scope.}

Syntax

call blacs_barrier( icontxt, scope )
Input Parameters

icontext  INTEGER. Integer handle that indicates the context.

scope  CHARACTER*1. Parameter that indicates whether a process row (scope='R'), column ('C'), or entire grid ('A') will participate in the barrier.

Description

This routine holds up execution of all processes within the indicated scope until they have all called the routine.

Examples of BLACS Routines Usage
Data Fitting Functions

Data Fitting functions in Intel® MKL provide spline-based interpolation capabilities that you can use to approximate functions, function derivatives or integrals, and perform cell search operations.

The Data Fitting component is task based. The task is a data structure or descriptor that holds the parameters related to a specific Data Fitting operation. You can modify the task parameters using the task editing functionality of the library.

For definition of the implemented operations, see Mathematical Conventions.

Data Fitting routines use the following workflow to process a task:

1. Create a task or multiple tasks.
2. Modify the task parameters.
3. Perform a Data Fitting computation.
4. Destroy the task or tasks.

All Data Fitting functions fall into the following categories:

- **Task Creation and Initialization Routines** - routines that create a new Data Fitting task descriptor and initialize the most common parameters, such as partition of the interpolation interval, values of the vector-valued function, and the parameters describing their structure.

- **Task Configuration Routines** - routines that set, modify, or query parameters in an existing Data Fitting task.

- **Computational Routines** - routines that perform Data Fitting computations, such as construction of a spline, interpolation, computation of derivatives and integrals, and search.

- **Task Destructors** - routines that delete Data Fitting task descriptors and deallocate resources.

You can access the Data Fitting routines through the Fortran and C89/C99 language interfaces. You can also use the C89 interface with more recent versions of C/C++, or the Fortran 90 interface with programs written in Fortran 95.

The `$(MKL)/include` directory of the Intel® MKL contains the following Data Fitting header files:

- mkl_df.h

You can find examples that demonstrate usage of Data Fitting routines in the `$(MKL)/examples/datafittingc` directory.

Data Fitting Function Naming Conventions

The interface of the Data Fitting functions, types, and constants are case-sensitive and can be in lowercase, uppercase, and mixed case.

The names of all routines have the following structure:

`df[datatype]<base_name>

where

- `df` is a prefix indicating that the routine belongs to the Data Fitting component of Intel® MKL.
- `[datatype]` field specifies the type of the input and/or output data and can be `s` (for the single precision real type), `d` (for the double precision real type), or `i` (for the integer type). This field is omitted in the names of the routines that are not data type dependent.
- `<base_name>` field specifies the functionality the routine performs. For example, this field can be `NewTask1D`, `Interpolate1D`, or `DeleteTask`
Data Fitting Functions

Data Fitting Function Data Types

The Data Fitting component provides routines for processing single and double precision real data types. The results of cell search operations are returned as a generic integer data type.

All Data Fitting routines use the following data type:

<table>
<thead>
<tr>
<th>Type</th>
<th>Data Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFTaskPtr</td>
<td>Pointer to a task</td>
</tr>
</tbody>
</table>

**NOTE**
The actual size of the generic integer type is platform-dependent. Before compiling your application, you need to set an appropriate byte size for integers. For details, see section Using the ILP64 Interface vs. LP64 Interface of the Intel® MKL Developer Guide.

Mathematical Conventions for Data Fitting Functions

This section explains the notation used for Data Fitting function descriptions. Spline notations are based on the terminology and definitions of [deBoor2001]. The Subbotin quadratic spline definition follows the conventions of [StechSub76]. The quasi-uniform partition definition is based on [Schumaker2007].

### Mathematical Notation in the Data Fitting Component

<table>
<thead>
<tr>
<th>Concept</th>
<th>Mathematical Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition of interpolation interval ([a, b]), where (x_i) denotes breakpoints. ([x_i, x_{i+1})) denotes a sub-interval (cell) of size (\Delta_i = x_{i+1} - x_i).</td>
<td>({x_i}_{i=1,...,n}), where (a = x_1 &lt; x_2 &lt; ... &lt; x_n = b)</td>
</tr>
<tr>
<td>Quasi-uniform partition of interpolation interval ([a, b])</td>
<td>Partition ({x_i}<em>{i=1,...,n}) which meets the constraint with a constant (C) defined as (1 \leq M/ m \leq C), where (M = \max</em>{i=1,...,n-1} (\Delta_i)), (m = \min_{i=1,...,n-1} (\Delta_i)), (\Delta_i = x_{i+1} - x_i).</td>
</tr>
<tr>
<td>Vector-valued function of dimension (p) being fit</td>
<td>(f(x) = (f_1(x), ..., f_p(x)))</td>
</tr>
<tr>
<td>Piecewise polynomial (PP) function (f) of order (k+1)</td>
<td>(f(x) = P_i(x), \text{ if } x \in [x_i, x_{i+1}), i = 1, ..., n-1) where ({x_i}<em>{i=1,...,n}) is a strictly increasing sequence of breakpoints. (P_i(x) = c</em>{i,0} + c_{i,1}(x - x_i) + ... + c_{i,k}(x - x_i)^k) is a polynomial of degree (k) (order (k+1)) over the interval (x \in [x_i, x_{i+1})).</td>
</tr>
<tr>
<td>Function (p) agrees with function (f) at the points ({x_i}_{i=1,...,n}).</td>
<td>For every point (\zeta) in sequence ({x_i}_{i=1,...,n}) that occurs (m) times, the equality (p^{(i-1)}(\zeta) = f^{(i-1)}(\zeta)) holds for all (i = 1, ..., m), where (p^{(0)}(t)) is the derivative of the (i)-th order.</td>
</tr>
</tbody>
</table>
The $k$-th divided difference of function $f$ at points $x_i, \ldots, x_i + k$. This difference is the leading coefficient of the polynomial of order $k+1$ that agrees with $f$ at $x_i, \ldots, x_i + k$.

A $k$-order derivative of interpolant $f(x)$ at interpolation site $\tau$.

### Interpolants to the Function $f$ at $x_1, \ldots, x_n$ and Boundary Conditions

**Concept**

| Linear interpolant | $P_i(x) = c_{1,i} + c_{2,i}(x - x_i)$, where
| | $\bullet x \in [x_i, x_{i+1})$
| | $\bullet c_{1,i} = f(x_i)$
| | $\bullet c_{2,i} = [x_i, x_{i+1}]f$
| | $\bullet i = 1, \ldots, n-1$
| Piecewise parabolic interpolant | $P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2$, $x \in [x_i, x_{i+1})$
| | Coefficients $c_{1,i}, c_{2,i},$ and $c_{3,i}$ depend on the conditions:
| | $\bullet P_i(x_i) = f(x_i)$
| | $\bullet P_i(x_{i+1}) = f(x_{i+1})$
| | $\bullet P_i((x_{i+1} - x_i) / 2) = v_{i+1}$
| | where parameter $v_{i+1}$ depends on the interpolant being continuously differentiable:
| | $P_i^{(1)}(x_i) = P_i^{(1)}(x)$
| Piecewise parabolic Subbotin interpolant | $P(x) = P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + d_{3,i}((x - t_i) + 1)^2$
| | where
| | $\bullet x \in [t_i, t_{i+1})$
| | $\{t_i\}_{i=1, \ldots, n+1}$ is a sequence of knots such that
| | $\bullet t_1 = x_1, t_{n+1} = x_n$
| | $\bullet t_i \in (x_{i-1}, x_i), i = 2, \ldots, n$
| | $\bullet x^* = f(x) =\begin{cases}0, &\text{if } x < 0 \\
\quad x, &\text{if } x \geq 0\end{cases}$
| | Coefficients $c_{1,i}, c_{2,i}, c_{3,i},$ and $d_{3,i}$ depend on the following conditions:
| | $\bullet P_i(x_i) = f(x_i), P_i(x_{i+1}) = f(x_{i+1})$
| | $\bullet P(x)$ is a continuously differentiable polynomial of the second degree on $[t_i, t_{i+1}), i = 1, \ldots, n$.
| Piecewise cubic Hermite interpolant | $P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3$
| | where
| | $\bullet x \in [x_i, x_{i+1})$
| | $\bullet c_{1,i} = f(x_i)$
| | $\bullet c_{2,i} = s_i$
| | $\bullet c_{3,i} = ([x_i, x_{i+1}]f - s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i)$
| | $\bullet c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1}]) / (\Delta x_i)^2$
| | $\bullet i = 1, \ldots, n-1$
Piecewise cubic Bessel interpolant

\[ P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3, \]

where

- \( x \in [x_i, x_{i+1}) \)
- \( c_{1,i} = f(x_i) \)
- \( c_{2,i} = s_i \)
- \( c_{3,i} = ([x_i, x_{i+1}) \cdot s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i) \)
- \( c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1})) / (\Delta x_i)^2 \)
- \( i = 1, \ldots, n-1 \)
- \( s_i = (\Delta x_i[x_{i-1}, x_i] / \Delta x_{i-1}[x_i, x_{i+1})] / (\Delta x_i + \Delta x_{i+1}) \)

Piecewise cubic Akima interpolant

\[ P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3, \]

where

- \( x \in [x_i, x_{i+1}) \)
- \( c_{1,i} = f(x_i) \)
- \( c_{2,i} = s_i \)
- \( c_{3,i} = ([x_i, x_{i+1}) \cdot s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i) \)
- \( c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1})) / (\Delta x_i)^2 \)
- \( i = 1, \ldots, n-1 \)
- \( s_i = (w_{i+1}[x_{i-1}, x_i] / w_{i-1}[x_{i-1}, x_{i+1})] / (w_{i+1} + w_{i-1}), \)

where

\[ w_i = |[x_i, x_{i+1})/ - [x_{i-1}, x_i)| \]

Piecewise natural cubic interpolant

\[ P_i(x) = c_{1,i} + c_{2,i}(x - x_i) + c_{3,i}(x - x_i)^2 + c_{4,i}(x - x_i)^3, \]

where

- \( x \in [x_i, x_{i+1}) \)
- \( c_{1,i} = f(x_i) \)
- \( c_{2,i} = s_i \)
- \( c_{3,i} = ([x_i, x_{i+1}) \cdot s_i) / (\Delta x_i) - c_{4,i}(\Delta x_i) \)
- \( c_{4,i} = (s_i + s_{i+1} - 2[x_i, x_{i+1})] / (\Delta x_i)^2 \)
- \( i = 1, \ldots, n-1 \)
- Parameter \( s_i \) depends on the condition that the interpolant is twice continuously differentiable: \( P_{i-1}''(x_i) = P_i''(x_i). \)

Not-a-knot boundary condition. Parameters \( s_1 \) and \( s_n \) provide \( P_1 = P_2 \) and \( P_{n-1} = P_n \), so that the first and the last interior breakpoints are inactive.

Free-end boundary condition.

\[ f''(x_1) = f''(x_n) = 0 \]

Look-up interpolator for discrete set of points \((x_1, y_1), \ldots, (x_n, y_n)\).

\[ y'(x) = \begin{cases} 
  y_1, & \text{if } z = x_1 \\
  y_2, & \text{if } z = x_2 \\
  \ldots \\
  y_n, & \text{if } z = x_n \\
  \text{error, otherwise} 
\end{cases} \]
### Concept

<table>
<thead>
<tr>
<th>Concept</th>
<th>Mathematical Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step-wise constant continuous right interpolator.</td>
<td>[ y(x) = \begin{cases} y_1, &amp; \text{if } x_1 \leq x &lt; x_2 \ y_2, &amp; \text{if } x_2 \leq x &lt; x_3 \ \vdots \ y_{n-1}, &amp; \text{if } x_{n-1} \leq x &lt; x_n \ y_n, &amp; \text{if } x = x_n \end{cases} ]</td>
</tr>
<tr>
<td>Step-wise constant continuous left interpolator.</td>
<td>[ y(x) = \begin{cases} y_1, &amp; \text{if } x = x_1 \ y_2, &amp; \text{if } x_1 &lt; x \leq x_2 \ y_3, &amp; \text{if } x_2 &lt; x \leq x_3 \ \vdots \ y_n, &amp; \text{if } x_{n-1} &lt; x \leq x_n \end{cases} ]</td>
</tr>
</tbody>
</table>

### Data Fitting Usage Model

Consider an algorithm that uses the Data Fitting functions. Typically, such algorithms consist of four steps or stages:

1. **Create a task.** You can call the Data Fitting function several times to create multiple tasks.
   ```c
   status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );
   ```

2. **Modify the task parameters.**
   ```c
   status = dfdEditPPSpline1D( task, s_order, c_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint );
   ```

3. **Perform Data Fitting spline-based computations.** You may reiterate steps 2-3 as needed.
   ```c
   status = dfdInterpolate1D(task, estimate, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell );
   ```

4. **Destroy the task or tasks.**
   ```c
   status = dfDeleteTask( &task );
   ```

**See Also**

Data Fitting Usage Examples

### Data Fitting Usage Examples

The examples below illustrate several operations that you can perform with Data Fitting routines.

You can get source code for similar examples in the `.\examples\datafittingc` subdirectory of the Intel® MKL installation directory.

The following example demonstrates the construction of a linear spline using Data Fitting routines. The spline approximates a scalar function defined on non-uniform partition. The coefficients of the spline are returned as a one-dimensional array:

**Example of Linear Spline Construction**

```c
#include "mkl.h"
#define N 500 /* Size of partition, number of breakpoints */
#define SPLINE_ORDER DF_PP_LINEAR /* Linear spline to construct */
```
int main()
{
    int status; /* Status of a Data Fitting operation */
    DFTaskPtr task; /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx; /* The size of partition x */
    double x[N]; /* Partition x */
    MKL_INT xhint; /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny; /* Function dimension */
    double y[N]; /* Function values at the breakpoints */
    MKL_INT yhint; /* Additional information about the function */

    /* Parameters describing the spline */
    MKL_INT s_order; /* Spline order */
    MKL_INT s_type; /* Spline type */
    MKL_INT ic_type; /* Type of internal conditions */
    double* ic; /* Array of internal conditions */
    MKL_INT bc_type; /* Type of boundary conditions */
    double* bc; /* Array of boundary conditions */

    double scoeff[(N-1)* SPLINE_ORDER]; /* Array of spline coefficients */
    MKL_INT scoeffhint; /* Additional information about the coefficients */

    /* Initialize the partition */
    nx = N;
    /* Set values of partition x */
    ...
    xhint = DF_NO_HINT; /* No additional information about the function is provided. By default, the partition is non-uniform. */

    /* Initialize the function */
    ny = 1; /* The function is scalar. */
    /* Set function values */
    ...
    yhint = DF_NO_HINT; /* No additional information about the function is provided. */

    /* Create a Data Fitting task */
    status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );

    /* Check the Data Fitting operation status */
    ...

    /* Initialize spline parameters */
    s_order = DF_PP_LINEAR; /* Spline is of the second order. */
    s_type = DF_PP_DEFAULT; /* Spline is of the default type. */

    /* Define internal conditions for linear spline construction (none in this example) */
    ic_type = DF_NO_IC;
    ic = NULL;

    /* Define boundary conditions for linear spline construction (none in this example) */
    bc_type = DF_NO_BC;
    bc = NULL;
    scoeffhint = DF_NO_HINT; /* No additional information about the spline. */
The following example demonstrates cubic spline-based interpolation using Data Fitting routines. In this example, a scalar function defined on non-uniform partition is approximated by Bessel cubic spline using not-a-knot boundary conditions. Once the spline is constructed, you can use the spline to compute spline values at the given sites. Computation results are packed by the Data Fitting routine in row-major format.

**Example of Cubic Spline-Based Interpolation**

```c
#include "mkl.h"

#define NX 100 /* Size of partition, number of breakpoints */
#define NSITE 1000 /* Number of interpolation sites */
#define SPLINE_ORDER DF_PP_CUBIC /* A cubic spline to construct */

int main()
{
    int status; /* Status of a Data Fitting operation */
    DFTaskPtr task; /* Data Fitting operations are task based */

    /* Parameters describing the partition */
    MKL_INT nx; /* The size of partition x */
    double x[NX]; /* Partition x */
    MKL_INT xhint; /* Additional information about the structure of breakpoints */

    /* Parameters describing the function */
    MKL_INT ny; /* Function dimension */
    double y[NX]; /* Function values at the breakpoints */
    MKL_INT yhint; /* Additional information about the function */

    /* Parameters describing the spline */
    MKL_INT s_order; /* Spline order */
    MKL_INT s_type; /* Spline type */
    MKL_INT ic_type; /* Type of internal conditions */
```
double* ic;         /* Array of internal conditions */
MKL_INT  bc_type;    /* Type of boundary conditions */
double* bc;         /* Array of boundary conditions */

double scoeff[(NX-1)* SPLINE_ORDER];   /* Array of spline coefficients */
MKL_INT scoeffhint;            /* Additional information about the coefficients */

/* Parameters describing interpolation computations */
MKL_INT nsite;        /* Number of interpolation sites */
double site[NSITE];   /* Array of interpolation sites */
MKL_INT sitehint;     /* Additional information about the structure of
interpolation sites */

MKL_INT ndorder, dorder;    /* Parameters defining the type of interpolation */
double* datahint;   /* Additional information on partition and interpolation sites */
double r[NSITE];    /* Array of interpolation results */
MKL_INT rhint;     /* Additional information on the structure of the results */
MKL_INT* cell;      /* Array of cell indices */

/* Initialize the partition */
xhint = DF_NON_UNIFORM_PARTITION;  /* The partition is non-uniform. */

/* Initialize the function */
ny = 1;               /* The function is scalar. */

/* Set function values */

/* Create a Data Fitting task */
status = dfdNewTask1D( &task, nx, x, xhint, ny, y, yhint );

/* Check the Data Fitting operation status */

/* Initialize spline parameters */
s_order = DF_PP_CUBIC;     /* Spline is of the fourth order (cubic spline). */
s_type = DF_PP_BESSEL;     /* Spline is of the Bessel cubic type. */

/* Define internal conditions for cubic spline construction (none in this example) */
ic_type = DF_NO_IC;
ic = NULL;

/* Use not-a-knot boundary conditions. In this case, the is first and the last
interior breakpoints are inactive, no additional values are provided. */
bc_type = DF_BC_NOT_A_KNOT;
bc = NULL;
scoeffhint = DF_NO_HINT;    /* No additional information about the spline. */

/* Set spline parameters in the Data Fitting task */
status = dfdEditPPSpline1D( task, s_order, s_type, bc_type, bc, ic_type,
ic, scoeff, scoeffhint );
/* Check the Data Fitting operation status */
...

/* Use a standard method to construct a cubic Bessel spline: */
/* $P_i(x) = c_{i,0} + c_{i,1}(x - x_i) + c_{i,2}(x - x_i)^2 + c_{i,3}(x - x_i)^3$, */
/* The library packs spline coefficients to array scoeff: */
/* scoeff[4*i+0] = c_{i,0}, scoef[4*i+1] = c_{i,1}, */
/* scoeff[4*i+2] = c_{i,2}, scoef[4*i+1] = c_{i,3}, */
/* i=0,...,N-2 */
status = dfdConstruct1D( task, DF_PP_SPLINE, DF_METHOD_STD );

/* Check the Data Fitting operation status */
...

/* Initialize interpolation parameters */
nsite = NSITE;

/* Set site values */
...

sitehint = DF_NON_UNIFORM_PARTITION; /* Partition of sites is non-uniform */

/* Request to compute spline values */
ndorder = 1;
dorder = 1;
datahint = DF_NO_APRIOI_INFO; /* No additional information about breakpoints or 
sites is provided. */
rhint = DF_MATRIX_STORAGE_ROWS; /* The library packs interpolation results 
in row-major format. */
cell = NULL; /* Cell indices are not required. */

/* Solve interpolation problem using the default method: compute the spline values 
at the points site(i), i=0,..., nsite-1 and place the results to array r */
status = dfdInterpolate1D( task, DF_INTERP, DF_METHOD_PP, nsite, site, 
    sitehint, ndorder, &dorder, datahint, r, rhint, cell );

/* Check Data Fitting operation status */
...

/* De-allocate Data Fitting task resources */
status = dfDeleteTask( &task );
/* Check Data Fitting operation status */
...
return 0;
}

The following example demonstrates how to compute indices of cells containing given sites. This example 
uses uniform partition presented with two boundary points. The sites are in the ascending order.

**Example of Cell Search**

```c
#include "mkl.h"
#define NX 100 /* Size of partition, number of breakpoints */
#define NSITE 1000 /* Number of interpolation sites */

int main()
{ 
```
```c
int status;           /* Status of a Data Fitting operation */
DFTaskPtr task;      /* Data Fitting operations are task based */

/* Parameters describing the partition */
MKL_INT nx;          /* The size of partition x */
float x[2];          /* Partition x is uniform and holds endpoints
                     of interpolation interval [a, b] */
MKL_INT xhint;       /* Additional information about the structure of breakpoints */

/* Parameters describing the function */
MKL_INT ny;          /* Function dimension */
float *y;            /* Function values at the breakpoints */
MKL_INT yhint;       /* Additional information about the function */

/* Parameters describing cell search */
MKL_INT nsite;       /* Number of interpolation sites */
float site[NSITE];   /* Array of interpolation sites */
MKL_INT sitehint;    /* Additional information about the structure of sites */
float* datahint;     /* Additional information on partition and interpolation sites */

MKL_INT cell[NSITE]; /* Array for cell indices */

/* Initialize a uniform partition */
x = NX;
/* Set values of partition x: for uniform partition,           */
/* provide end-points of the interpolation interval [-1.0,1.0] */
x[0] = -1.0f; x[1] = 1.0f;
xhint = DF_UNIFORM_PARTITION; /* Partition is uniform */

/* Initialize function parameters */
/* In cell search, function values are not necessary and are set to zero/NUL values */
ny = 0;
y = NULL;
yhint = DF_NO_HINT;

/* Create a Data Fitting task */
status = dfsNewTask1D( &task, nx, x, xhint, ny, y, yhint );

/* Check Data Fitting operation status */
...

/* Initialize interpolation (cell search) parameters */
nsite = NSITE;

/* Set sites in the ascending order */
...
sitehint = DF_SORTED_DATA; /* Sites are provided in the ascending order. */
datahint = DF_NO_APRIORI_INFO; /* No additional information
                                 about breakpoints/sites is provided.*/

/* Use a standard method to compute indices of the cells that contain
   interpolation sites. The library places the index of the cell containing
   site(i) to the cell(i), i=0,...,nsite-1 */
status = dfsSearchCells1D( task, DF_METHOD_STD, nsite, site, sitehint, datahint, cell );
/* Check Data Fitting operation status */
...
```
Data Fitting Function Task Status and Error Reporting

The Data Fitting routines report a task status through integer values. Negative status values indicate errors, while positive values indicate warnings. An error can be caused by invalid parameter values or a memory allocation failure.

The status codes have symbolic names predefined in the header file as macros via the `#define` statements.

If no error occurred, the function returns the `DF_STATUS_OK` code defined as zero:

```c
#define DF_STATUS_OK 0
```

In case of an error, the function returns a non-zero error code that specifies the origin of the failure. Header files define the following status codes:

### Status Codes in the Data Fitting Component

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Common Status Codes</strong></td>
<td></td>
</tr>
<tr>
<td><code>DF_STATUS_OK</code></td>
<td>Operation completed successfully.</td>
</tr>
<tr>
<td><code>DF_ERROR_NULL_TASK</code></td>
<td>Data Fitting task is a <code>NULL</code> pointer.</td>
</tr>
<tr>
<td><code>DF_ERROR_MEM_FAILURE</code></td>
<td>Memory allocation failure.</td>
</tr>
<tr>
<td><code>DF_ERROR_METHOD_NOT_SUPPORTED</code></td>
<td>Requested method is not supported.</td>
</tr>
<tr>
<td><code>DF_ERROR_COMP_TYPE_NOT_SUPPORTED</code></td>
<td>Requested computation type is not supported.</td>
</tr>
<tr>
<td><code>DF_ERROR_NULL_PTR</code></td>
<td>Pointer to parameter is null.</td>
</tr>
<tr>
<td><strong>Data Fitting Task Creation and Initialization, and Generic Editing Operations</strong></td>
<td></td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_NX</code></td>
<td>Invalid number of breakpoints.</td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_X</code></td>
<td>Array of breakpoints is invalid.</td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_X_HINT</code></td>
<td>Invalid hint describing the structure of the partition.</td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_NY</code></td>
<td>Invalid dimension of vector-valued function $y$.</td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_Y</code></td>
<td>Array of function values is invalid.</td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_Y_HINT</code></td>
<td>Invalid flag describing the structure of function $y$.</td>
</tr>
<tr>
<td><strong>Data Fitting Task-Specific Editing Operations</strong></td>
<td></td>
</tr>
<tr>
<td><code>DF_ERROR_BAD_SPLINE_ORDER</code></td>
<td>Invalid spline order.</td>
</tr>
</tbody>
</table>
### Status Code

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_ERROR_BAD_SPLINE_TYPE</td>
<td>Invalid spline type.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_IC_TYPE</td>
<td>Type of internal conditions used for spline construction is invalid.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_IC</td>
<td>Array of internal conditions for spline construction is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_BC_TYPE</td>
<td>Type of boundary conditions used in spline construction is invalid.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_BC</td>
<td>Array of boundary conditions for spline construction is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_PP_COEFF</td>
<td>Array of piecewise polynomial spline coefficients is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_PP_COEFF_HINT</td>
<td>Invalid flag describing the structure of the piecewise polynomial spline coefficients.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_PERIODIC_VAL</td>
<td>Function values at the endpoints of the interpolation interval are not equal as required in periodic boundary conditions.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_DATA_ATTR</td>
<td>Invalid attribute of the pointer to be set or modified in Data Fitting task descriptor with the df? EditIdxPtr task editor.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_DATA_IDX</td>
<td>Index of the pointer to be set or modified in the Data Fitting task descriptor with the df? EditIdxPtr task editor is out of the pre-defined range.</td>
</tr>
</tbody>
</table>

### Data Fitting Computation Operations

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_ERROR_BAD_NSITE</td>
<td>Invalid number of interpolation sites.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_SITE</td>
<td>Array of interpolation sites is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_SITE_HINT</td>
<td>Invalid flag describing the structure of interpolation sites.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_NDORDER</td>
<td>Invalid size of the array defining derivative orders to be computed at interpolation sites.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_DORDER</td>
<td>Array defining derivative orders to be computed at interpolation sites is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_DATA_HINT</td>
<td>Invalid flag providing additional information about partition or interpolation sites.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_INTERP</td>
<td>Array of spline-based interpolation results is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_INTERP_HINT</td>
<td>Invalid flag defining the structure of spline-based interpolation results.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_CELL_IDX</td>
<td>Array of indices of partition cells containing interpolation sites is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_NLIM</td>
<td>Invalid size of arrays containing integration limits.</td>
</tr>
</tbody>
</table>
### Status Code

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_ERROR_BAD_LLIM</td>
<td>Array of the left-side integration limits is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_RLIM</td>
<td>Array of the right-side integration limits is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_INTEGR</td>
<td>Array of spline-based integration results is not defined.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_INTEGR_HINT</td>
<td>Invalid flag providing the structure of the array of spline-based integration results.</td>
</tr>
<tr>
<td>DF_ERROR_BAD_LOOKUP_INTERP_SITE</td>
<td>Bad site provided for interpolation with look-up interpolator.</td>
</tr>
</tbody>
</table>

**NOTE**

The routine that estimates piecewise polynomial cubic spline coefficients can return internal error codes related to the specifics of the implementation. Such error codes indicate invalid input data or other issues unrelated to Data Fitting routines.

---

### Data Fitting Task Creation and Initialization Routines

Task creation and initialization routines are functions used to create a new task descriptor and initialize its parameters. The Data Fitting component provides the `dfNewTask1D` routine that creates and initializes a new task descriptor for a one-dimensional Data Fitting task.

**dfNewTask1D**

*Creates and initializes a new task descriptor for a one-dimensional Data Fitting task.*

**Syntax**

```c
status = dfsNewTask1D(&task, nx, x, xhint, ny, y, yhint)
status = dfdNewTask1D(&task, nx, x, xhint, ny, y, yhint)
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nx</td>
<td>const MKL_INT</td>
<td>Number of breakpoints representing partition of interpolation interval ([a,b]).</td>
</tr>
<tr>
<td>x</td>
<td>const float* for dfsNewTask1D</td>
<td>One-dimensional array containing the strictly sorted breakpoints from interpolation interval ([a,b]). The structure of the array is defined by parameter xhint.</td>
</tr>
</tbody>
</table>
Name  Type  Description

- If partition is non-uniform or quasi-uniform, the array should contain \( nx \) strictly ordered values.
- If partition is uniform, the array should contain two entries that represent endpoints of interpolation interval \([a, b]\).

**Caution**
The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct.

\[\begin{array}{ll}
xhint & \text{const MKL_INT} \\
ny & \text{const MKL_INT} \\
y & \text{const float* for dfsNewTask} \\
& \text{const double* for dfdNewTask} \\
yhint & \text{const MKL_INT} \\
\end{array}\]

*xhint* A flag describing the structure of partition \( x \). For the list of possible values of \( xhint \), see table "Hint Values for Partition \( x \)". If you set the flag to the DF_NO_HINT value, the library interprets the partition as non-uniform.

*ny* Dimension of vector-valued function \( y \).

*y* Vector-valued function \( y \), array of size \( nx \times ny \).

*yhint* A flag describing the structure of array \( y \). Valid hint values are listed in table "Hint Values for Vector-Valued Function \( y \)". If you set the flag to the DF_NO_HINT value, the library assumes that all \( ny \) coordinates of the vector-valued function \( y \) are provided and stored in row-major format.

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DF_STATUS_OK if the task is created successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Non-zero error code if the task creation failed. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

**Description**
The `df?NewTask1D` routine creates and initializes a new Data Fitting task descriptor with user-specified parameters for a one-dimensional Data Fitting task. The \( x \) and \( nx \) parameters representing the partition of interpolation interval \([a, b]\) are mandatory. If you provide invalid values for these parameters, such as a NULL pointer \( x \) or the number of breakpoints smaller than two, the routine does not create the Data Fitting task and returns an error code.

If you provide a vector-valued function \( y \), make sure that the function dimension \( ny \) and the array of function values \( y \) are both valid. If any of these parameters are invalid, the routine does not create the Data Fitting task and returns an error code.
If you store coordinates of the vector-valued function $y$ in non-contiguous memory locations, you can set the $yhint$ flag to $DF_{1ST\_COORDINATE}$, and pass only the first coordinate of the function into the task creation routine. After successful creation of the Data Fitting task, you can pass the remaining coordinates using the $df?EditIdxPtr$ task editor.

If the routine fails to create the task descriptor, it returns a NULL task pointer.

The routine supports the following hint values for partition $x$:

**Hint Values for Partition $x$**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DF_NON_UNIFORM_PARTITION$</td>
<td>Partition is non-uniform.</td>
</tr>
<tr>
<td>$DF_QUASI_UNIFORM_PARTITION$</td>
<td>Partition is quasi-uniform.</td>
</tr>
<tr>
<td>$DF_UNIFORM_PARTITION$</td>
<td>Partition is uniform.</td>
</tr>
<tr>
<td>$DF_NO_HINT$</td>
<td>No hint is provided. By default, partition is interpreted as non-uniform.</td>
</tr>
</tbody>
</table>

The routine supports the following hint values for the vector-valued function:

**Hint Values for Vector-Valued Function $y$**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DF_MATRIX_STORAGE_ROWS$</td>
<td>Data is stored in row-major format according to C conventions.</td>
</tr>
<tr>
<td>$DF_MATRIX_STORAGE_COLS$</td>
<td>Data is stored in column-major format according to Fortran conventions.</td>
</tr>
<tr>
<td>$DF_1ST_COORDINATE$</td>
<td>The first coordinate of vector-valued data is provided.</td>
</tr>
<tr>
<td>$DF_NO_HINT$</td>
<td>No hint is provided. By default, the coordinates of vector-valued function $y$ are provided and stored in row-major format.</td>
</tr>
</tbody>
</table>

**NOTE**

You must preserve the arrays $x$ (breakpoints) and $y$ (vector-valued functions) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

**Task Configuration Routines**

In order to configure tasks, you can use task editors and task query routines.

Task editors initialize or change the predefined Data Fitting task parameters. You can use task editors to initialize or modify pointers to arrays or parameter values.

Task editors can be task-specific or generic. Task-specific editors can modify more than one parameter related to a specific task. Generic editors modify a single parameter at a time.

The Data Fitting component of Intel® MKL provides the following task editors:
**Data Fitting Task Editors**

<table>
<thead>
<tr>
<th>Editor</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>df?EditPPSpline1D</td>
<td>Changes parameters of the piecewise polynomial spline.</td>
<td>Task-specific</td>
</tr>
<tr>
<td>df?EditPtr</td>
<td>Changes a pointer in the task descriptor.</td>
<td>Generic</td>
</tr>
<tr>
<td>dfiEditVal</td>
<td>Changes a value in the task descriptor.</td>
<td>Generic</td>
</tr>
<tr>
<td>df?EditIdxPtr</td>
<td>Changes a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.</td>
<td>Generic</td>
</tr>
</tbody>
</table>

Task query routines are used to read the predefined Data Fitting task parameters. You can use task query routines to read the values of pointers or parameters.

Task query routines are generic (not task-specific), allowing you to read a single parameter at a time.

The Data Fitting component of the Intel® MKL provides the following task query routines:

**Data Fitting Task Query Routines**

<table>
<thead>
<tr>
<th>Editor</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>df?QueryPtr</td>
<td>Queries a pointer in the task descriptor.</td>
<td>Generic</td>
</tr>
<tr>
<td>dfiQueryVal</td>
<td>Queries a value in the task descriptor.</td>
<td>Generic</td>
</tr>
<tr>
<td>df?QueryIdxPtr</td>
<td>Queries a coordinate of data represented in matrix format, such as a vector-valued function or spline coefficients.</td>
<td>Generic</td>
</tr>
</tbody>
</table>

### df?EditPPSpline1D

**Modifies parameters representing a spline in a Data Fitting task descriptor.**

**Syntax**

```c
status = dfsEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

```c
status = dfdEditPPSpline1D(task, s_order, s_type, bc_type, bc, ic_type, ic, scoeff, scoeffhint)
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>s_order</td>
<td>const MKL_INT</td>
<td>Spline order. The parameter takes one of the values described in table &quot;Spline Orders Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>s_type</td>
<td>const MKL_INT</td>
<td>Spline type. The parameter takes one of the values described in table &quot;Spline Types Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>bc_type</td>
<td>const MKL_INT</td>
<td>Type of boundary conditions. The parameter takes one of the values described in table &quot;Boundary Conditions Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>bc</td>
<td>const float* for</td>
<td>Pointer to boundary conditions. The size of the array is defined by the value of parameter bc_type:</td>
</tr>
<tr>
<td></td>
<td>dfsEditPPSpline1D</td>
<td>• If you set free-end or not-a-knot boundary conditions, pass the NULL pointer to this parameter.</td>
</tr>
<tr>
<td></td>
<td>const double* for</td>
<td>• If you combine boundary conditions at the endpoints of the interpolation interval, pass an array of two elements.</td>
</tr>
<tr>
<td></td>
<td>dfdEditPPSpline1D</td>
<td>• If you set a boundary condition for the default quadratic spline or a periodic condition for Hermite or the default cubic spline, pass an array of one element.</td>
</tr>
<tr>
<td>ic_type</td>
<td>const MKL_INT</td>
<td>Type of internal conditions. The parameter takes one of the values described in table &quot;Internal Conditions Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>ic</td>
<td>const float* for</td>
<td>A non-NULL pointer to the array of internal conditions. The size of the array is defined by the value of parameter ic_type:</td>
</tr>
<tr>
<td></td>
<td>dfsEditPPSpline1D</td>
<td>• If you set first derivatives or second derivatives internal conditions (ic_type=DF_IC_1ST_DER or ic_type=DF_IC_2ND_DER), pass an array of n-1 derivative values at the internal points of the interpolation interval.</td>
</tr>
<tr>
<td></td>
<td>const double* for</td>
<td>• If you set the knot values internal condition for Subbotin spline (ic_type=DF_IC_Q_KNOT) and the knot partition is non-uniform, pass an array of n+1 elements.</td>
</tr>
<tr>
<td></td>
<td>dfdEditPPSpline1D</td>
<td>• If you set the knot values internal condition for Subbotin spline (ic_type=DF_IC_Q_KNOT) and the knot partition is uniform, pass an array of four elements.</td>
</tr>
<tr>
<td>sccoeff</td>
<td>const float* for</td>
<td>Spline coefficients. An array of size $ny \times s_order \times (nx-1)$. The storage format of the coefficients in the array is defined by the value of flag sccoeffhint.</td>
</tr>
<tr>
<td></td>
<td>dfsEditPPSpline1D</td>
<td>A flag describing the structure of the array of spline coefficients. For valid hint values, see table &quot;Hint Values for Spline Coefficients&quot;. The library stores the coefficients in row-major format. The default value is DF_NO_HINT.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
</tbody>
</table>

- DF_STATUS_OK if the routine execution completed successfully.
- Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.

Description

The editor modifies parameters that describe the order, type, boundary conditions, internal conditions, and coefficients of a spline. The spline order definition is provided in the "Mathematical Conventions" section. You can set the spline order to any value supported by Data Fitting functions. The table below lists the available values:

Spline Orders Supported by the Data Fitting Functions

<table>
<thead>
<tr>
<th>Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_PP_STD</td>
<td>Artificial value. Use this value for look-up and step-wise constant interpolants only.</td>
</tr>
<tr>
<td>DF_PP_LINEAR</td>
<td>Piecewise polynomial spline of the second order (linear spline).</td>
</tr>
<tr>
<td>DF_PP_QUADRATIC</td>
<td>Piecewise polynomial spline of the third order (quadratic spline).</td>
</tr>
<tr>
<td>DF_PP_CUBIC</td>
<td>Piecewise polynomial spline of the fourth order (cubic spline).</td>
</tr>
</tbody>
</table>

To perform computations with a spline not supported by Data Fitting routines, set the parameter defining the spline order and pass the spline coefficients to the library in the supported format. For format description, see figure "Row-major Coefficient Storage Format".

The table below lists the supported spline types:

Spline Types Supported by Data Fitting Functions

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_PP_DEFAULT</td>
<td>The default spline type. You can use this type with linear, quadratic, or user-defined splines.</td>
</tr>
<tr>
<td>DF_PP_SUBBOTIN</td>
<td>Quadratic splines based on Subbotin algorithm, [StechSub76].</td>
</tr>
<tr>
<td>DF_PP_NATURAL</td>
<td>Natural cubic spline.</td>
</tr>
<tr>
<td>DF_PP_HERMITE</td>
<td>Hermite cubic spline.</td>
</tr>
<tr>
<td>DF_PP_BESSEL</td>
<td>Bessel cubic spline.</td>
</tr>
<tr>
<td>DF_PP_AKIMA</td>
<td>Akima cubic spline.</td>
</tr>
<tr>
<td>DF_LOOKUP_INTERPOLANT</td>
<td>Look-up interpolant.</td>
</tr>
</tbody>
</table>
If you perform computations with look-up or step-wise constant interpolants, set the spline order to the `DF_PP_STD` value.

Construction of specific splines may require boundary or internal conditions. To compute coefficients of such splines, you should pass boundary or internal conditions to the library by specifying the type of the conditions and providing the necessary values. For splines that do not require additional conditions, such as linear splines, set condition types to `DF_NO_BC` and `DF_NO_IC`, and pass `NULL` pointers to the conditions. The table below defines the supported boundary conditions:

**Boundary Conditions Supported by Data Fitting Functions**

<table>
<thead>
<tr>
<th>Boundary Condition</th>
<th>Description</th>
<th>Spline</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DF_NO_BC</code></td>
<td>No boundary conditions provided.</td>
<td>All</td>
</tr>
<tr>
<td><code>DF_BC_NOT_A_KNOT</code></td>
<td>Not-a-knot boundary conditions.</td>
<td>Akima, Bessel, Hermite, natural cubic</td>
</tr>
<tr>
<td><code>DF_BC_FREE_END</code></td>
<td>Free-end boundary conditions.</td>
<td>Akima, Bessel, Hermite, natural cubic, quadratic Subbotin</td>
</tr>
<tr>
<td><code>DF_BC_1ST_LEFT_DER</code></td>
<td>The first derivative at the left endpoint.</td>
<td>Akima, Bessel, Hermite, natural cubic, quadratic Subbotin</td>
</tr>
<tr>
<td><code>DF_BC_1ST_RIGHT_DER</code></td>
<td>The first derivative at the right endpoint.</td>
<td>Akima, Bessel, Hermite, natural cubic, quadratic Subbotin</td>
</tr>
<tr>
<td><code>DF_BC_2ST_LEFT_DER</code></td>
<td>The second derivative at the left endpoint.</td>
<td>Akima, Bessel, Hermite, natural cubic, quadratic Subbotin</td>
</tr>
<tr>
<td><code>DF_BC_2ND_RIGHT_DER</code></td>
<td>The second derivative at the right endpoint.</td>
<td>Akima, Bessel, Hermite, natural cubic, quadratic Subbotin</td>
</tr>
<tr>
<td><code>DF_BC_PERIODIC</code></td>
<td>Periodic boundary conditions.</td>
<td>Linear, all cubic splines</td>
</tr>
<tr>
<td><code>DF_BC_Q_VAL</code></td>
<td>Function value at point $(x_0 + x_1)/2$</td>
<td>Default quadratic</td>
</tr>
</tbody>
</table>
NOTE
To construct a natural cubic spline, pass these settings to the editor:
• **DF_PP_CUBIC** as the spline order,
• **DF_PP_NATURAL** as the spline type, and
• **DF_BC_FREE_END** as the boundary condition.

To construct a cubic spline with other boundary conditions, pass these settings to the editor:
• **DF_PP_CUBIC** as the spline order,
• **DF_PP_NATURAL** as the spline type, and
• the required type of boundary condition.

For Akima, Hermite, Bessel, and default cubic splines use the corresponding type defined in Table Spline Types Supported by Data Fitting Functions.

You can combine the values of boundary conditions with a bitwise OR operation. This permits you to pass combinations of first and second derivatives at the endpoints of the interpolation interval into the library. To pass a first derivative at the left endpoint and a second derivative at the right endpoint, set the boundary conditions to **DF_BC_1ST_LEFT_DER** OR **DF_BC_2ND_RIGHT_DER**.

You should pass the combined boundary conditions as an array of two elements. The first entry of the array contains the value of the boundary condition for the left endpoint of the interpolation interval, and the second entry - for the right endpoint. Pass other boundary conditions as arrays of one element.

For the conditions defined as a combination of valid values, the library applies the following rules to identify the boundary condition type:
• If not required for spline construction, the value of boundary conditions is ignored.
• Not-a-knot condition has the highest priority. If set, other boundary conditions are ignored.
• Free-end condition has the second priority after the not-a-knot condition. If set, other boundary conditions are ignored.
• Periodic boundary condition has the next priority after the free-end condition.
• The first derivative has higher priority than the second derivative at the right and left endpoints.

If you set the periodic boundary condition, make sure that function values at the endpoints of the interpolation interval are identical. Otherwise, the library returns an error code. The table below specifies the values to be provided for each type of spline if the periodic boundary condition is set.

**Boundary Requirements for Periodic Conditions**

<table>
<thead>
<tr>
<th>Spline Type</th>
<th>Periodic Boundary Condition Support</th>
<th>Boundary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>Yes</td>
<td>Not required</td>
</tr>
<tr>
<td>Default quadratic</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Subbotin quadratic</td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>Natural cubic</td>
<td>Yes</td>
<td>Not required</td>
</tr>
<tr>
<td>Bessel</td>
<td>Yes</td>
<td>Not required</td>
</tr>
<tr>
<td>Akima</td>
<td>Yes</td>
<td>Not required</td>
</tr>
<tr>
<td>Hermite cubic</td>
<td>Yes</td>
<td>First derivative</td>
</tr>
<tr>
<td>Default cubic</td>
<td>Yes</td>
<td>Second derivative</td>
</tr>
</tbody>
</table>
Internal conditions supported in the Data Fitting domain that you can use for the \textit{ic\_type} parameter are the following:

**Internal Conditions Supported by Data Fitting Functions**

<table>
<thead>
<tr>
<th>Internal Condition</th>
<th>Description</th>
<th>Spline</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_NO_IC</td>
<td>No internal conditions provided.</td>
<td></td>
</tr>
<tr>
<td>DF_IC_1ST_DER</td>
<td>Array of first derivatives of size (n-2), where (n) is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.</td>
<td>Hermite cubic</td>
</tr>
<tr>
<td>DF_IC_2ND_DER</td>
<td>Array of second derivatives of size (n-2), where (n) is the number of breakpoints. Derivatives are applicable to each coordinate of the vector-valued function.</td>
<td>Default cubic</td>
</tr>
<tr>
<td>DF_IC_Q_KNOT</td>
<td>Knot array of size (n+1), where (n) is the number of breakpoints.</td>
<td>Subbotin quadratic</td>
</tr>
</tbody>
</table>

To construct a Subbotin quadratic spline, you have three options to get the array of knots in the library:

- If you do not provide the knots, the library uses the default values of knots \(t = \{t_i\}, i = 0, ..., n\) according to the rule:
  \[t_0 = x_0, t_n = x_{n-1}, t_i = (x_i + x_{i-1})/2, i = 1, ..., n - 1.\]
- If you provide the knots in an array of size \(n + 1\), the knots form a non-uniform partition. Make sure that the knot values you provide meet the following conditions:
  \[t_0 = x_0, t_n = x_{n-1}, t_i \in (x_{i-1}, x_i), i = 1, ..., n - 1.\]
- If you provide the knots in an array of size 4, the knots form a uniform partition
  \[t_0 = x_0, t_1 = l, t_2 = r, t_3 = x_{n-1}, \text{ where } l \in (x_0, x_1) \text{ and } r \in (x_{n-2}, x_{n-1}).\]

In this case, you need to set the value of the \textit{ic\_type} parameter holding the type of internal conditions to \texttt{DF\_IC\_Q\_KNOT OR DF\_UNIFORM\_PARTITION}.

**NOTE**
Since the partition is uniform, perform an OR operation with the \texttt{DF\_UNIFORM\_PARTITION} partition hint value described in Table Hint Values for Partition x.

For computations based on look-up and step-wise constant interpolants, you can avoid calling the \texttt{df?EditPPSpline1D} editor and directly call one of the routines for spline-based computation of spline values, derivatives, or integrals. For example, you can call the \texttt{df?Construct1D} routine to construct the required spline with the given attributes, such as order or type.

The memory location of the spline coefficients is defined by the \texttt{scoeff} parameter. Make sure that the size of the array is sufficient to hold \(n*y*s\_order* (nx-1)\) values.

The \texttt{df?EditPPSpline1D} routine supports the following hint values for spline coefficients:
## Hint Values for Spline Coefficients

<table>
<thead>
<tr>
<th>Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_1ST_COORDINATE</td>
<td>The first coordinate of vector-valued data is provided.</td>
</tr>
<tr>
<td>DF_NO_HINT</td>
<td>No hint is provided. By default, all sets of spline coefficients are stored in row-major format.</td>
</tr>
</tbody>
</table>

The coefficients for all coordinates of the vector-valued function are packed in memory one by one in successive order, from function $y_1$ to function $y_{ny}$.

Within each coordinate, the library stores the coefficients as an array, in row-major format:

$$c_1, 0, c_1, 1, \ldots, c_1, k, c_2, 0, c_2, 1, \ldots, c_2, k, \ldots, c_{n-1}, 0, c_{n-1}, 1, \ldots, c_{n-1}, k$$

Mapping of the coefficients to storage in the `scoeff` array is described below, where $c_{i,j}$ is the $j$th coefficient of the function

$$P_i(x) = c_{i,0} + c_{i,1}(x-x_i) + \ldots + c_{i,k}(x-x_i)^k.$$  

See Mathematical Conventions for more details on nomenclature and interpolants.

### Row-major Coefficient Storage Format

If you store splines corresponding to different coordinates of the vector-valued function at non-contiguous memory locations, do the following:

1. Set the `scoeffhint` flag to `DF_1ST_COORDINATE` and provide the spline for the first coordinate.
2. Pass the spline coefficients for the remaining coordinates into the Data Fitting task using the `df?EditIdxPtr` task editor.

Using the `df?EditPPSpline1D` task editor, you can provide to the Data Fitting task an already constructed spline that you want to use in computations. To ensure correct interpretation of the memory content, you should set the following parameters:

- Spline order and type, if appropriate. If the spline is not supported by the library, set the `s_type` parameter to `DF_PP_DEFAULT`.
- Pointer to the array of spline coefficients in row-major format.
• The `scoeffhint` parameter describing the structure of the array:
• Set the `scoeffhint` flag to the `DF_1ST_COORDINATE` value to pass spline coefficients stored at different memory locations. In this case, you can set the parameters that describe boundary and internal conditions to zero.
• Use the default value `DF_NO_HINT` for all other cases.

Before passing an already constructed spline into the library, you should call the `dfiEditVal` task editor to provide the dimension of the spline `DF_NY`. See table "Parameters Supported by the `dfiEditVal` Task Editor" for details.

After you provide the spline to the Data Fitting task, you can run computations that use this spline.

**NOTE**
You must preserve the arrays `bc` (boundary conditions), `ic` (internal conditions), and `scoeff` (spline coefficients) through the entire workflow of the Data Fitting computations for a task, as the task stores the addresses of the arrays for spline-based computations.

### `df?EditPtr`
Modifies a pointer to an array held in a Data Fitting task descriptor.

#### Syntax

```c
status = dfsEditPtr(task, ptr_attr, ptr)
status = dfdEditPtr(task, ptr_attr, ptr)
```

#### Include Files
- `mkl.h`

#### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task</code></td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td><code>ptr_attr</code></td>
<td>const MKL_INT</td>
<td>The parameter to change. For details, see the Pointer Attribute column in table &quot;Pointers Supported by the <code>df?EditPtr</code> Task Editor&quot;.</td>
</tr>
<tr>
<td><code>ptr</code></td>
<td>const float* for dfsEditPtr const double* for dfdEditPtr</td>
<td>New pointer. For details, see the Purpose column in table &quot;Pointers Supported by the <code>df?EditPtr</code> Task Editor&quot;.</td>
</tr>
</tbody>
</table>

#### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>status</code></td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>DF_STATUS_OK</code> if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code otherwise. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>
Description
The df?EditPtr editor replaces the pointer of type ptr_attr stored in a Data Fitting task descriptor with a new pointer ptr. The table below describes types of pointers supported by the editor:

**Pointers Supported by the df?EditPtr Task Editor**

<table>
<thead>
<tr>
<th>Pointer Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_X</td>
<td>Partition ( x ) of the interpolation interval, an array of strictly sorted breakpoints.</td>
</tr>
</tbody>
</table>

**Caution**
The array must be strictly sorted. If it is unordered, the results of data fitting routines are not correct.

<table>
<thead>
<tr>
<th>DF_Y</th>
<th>Vector-valued function ( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_IC</td>
<td>Internal conditions for spline construction. For details, see table &quot;Internal Conditions Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>DF_BC</td>
<td>Boundary conditions for spline construction. For details, see table &quot;Boundary Conditions Supported by Data Fitting Functions&quot;.</td>
</tr>
<tr>
<td>DF_PP_SCOEFF</td>
<td>Spline coefficients</td>
</tr>
</tbody>
</table>

You can use df?EditPtr to modify different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory. Use the df?EditIdxPtr editor if you need to modify pointers to coordinates of the vector-valued function or spline coefficients stored at non-contiguous memory locations.

If you modify a partition of the interpolation interval, then you should call the dfiEditVal task editor with the corresponding value of DF_XHINT, even if the structure of the partition remains the same.

If you pass a NULL pointer to the df?EditPtr task editor, the task remains unchanged and the routine returns an error code. For the predefined error codes, please see "Task Status and Error Reporting".

**NOTE**
You must preserve the arrays \( x \) (breakpoints), \( y \) (vector-valued functions), \( bc \) (boundary conditions), \( ic \) (internal conditions), and \( scoeff \) (spline coefficients) through the entire workflow of the Data Fitting computations which use those arrays, as the task stores the addresses of the arrays for spline-based computations.

dfiEditVal
*Modifies a parameter value in a Data Fitting task descriptor.*

**Syntax**

\[ status = dfiEditVal(task, val_attr, val) \]

**Include Files**
- mkl.h
Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>val_attr</td>
<td>const MKL_INT</td>
<td>The parameter to change. See table &quot;Parameters Supported by the dfiEditVal Task Editor&quot;.</td>
</tr>
<tr>
<td>val</td>
<td>const MKL_INT</td>
<td>A new parameter value. See table &quot;Parameters Supported by the dfiEditVal Task Editor&quot;.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code otherwise. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

Description

The dfiEditVal task editor replaces the parameter of type val_attr stored in a Data Fitting task descriptor with a new value val. The table below describes valid types of parameter val_attr supported by the editor:

Parameters Supported by the dfiEditVal Task Editor

<table>
<thead>
<tr>
<th>Parameter Attribute</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_NX</td>
<td>Number of breakpoints</td>
</tr>
<tr>
<td>DF_XHINT</td>
<td>A flag describing the structure of partition. See table &quot;Hint Values for Partition x&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_NY</td>
<td>Dimension of the vector-valued function</td>
</tr>
<tr>
<td>DF_YHINT</td>
<td>A flag describing the structure of the vector-valued function. See table &quot;Hint Values for Vector Function y&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_SPLINE_ORDER</td>
<td>Spline order. See table &quot;Spline Orders Supported by Data Fitting Functions&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_SPLINE_TYPE</td>
<td>Spline type. See table &quot;Spline Types Supported by Data Fitting Functions&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_BC_TYPE</td>
<td>Type of boundary conditions used in spline construction. See table &quot;Boundary Conditions Supported by Data Fitting Functions&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_IC_TYPE</td>
<td>Type of internal conditions used in spline construction. See table &quot;Internal Conditions Supported by Data Fitting Functions&quot; for the list of available values.</td>
</tr>
<tr>
<td>DF_PP_COEFF_HINT</td>
<td>A flag describing the structure of spline coefficients. See table &quot;Hint Values for Spline Coefficients&quot; for the list of available values.</td>
</tr>
</tbody>
</table>
Parameter Attribute | Purpose
--- | ---
DF_CHECK_FLAG | A flag which controls checking of Data Fitting parameters. See table "Possible Values for the DF_CHECK_FLAG Parameter" for the list of available values.

If you pass a zero value for the parameter describing the size of the arrays that hold coefficients for a partition, a vector-valued function, or a spline, the parameter held in the Data fitting task remains unchanged and the routine returns an error code. For the predefined error codes, see "Task Status and Error Reporting".

**Possible Values for the DF_CHECK_FLAG Parameter**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_ENABLE_CHECK_FLAG</td>
<td>Checks the correctness of parameters of Data Fitting computational routines (default mode).</td>
</tr>
<tr>
<td>DF_DISABLE_CHECK_FLAG</td>
<td>Disables checking of the correctness of parameters of Data Fitting computational routines.</td>
</tr>
</tbody>
</table>

Use DF_CHECK_FLAG for `val_attr` in order to control validation of parameters of Data Fitting computational routines such as `df?Construct1D`, `df?Interpolate1D/df?InterpolateEx1D`, and `df?SearchCells1D/df?SearchCellsEx1D`, which can perform better with a small number of interpolation sites or integration limits (fewer than one dozen). The default mode, with checking of parameters enabled, should be used as you develop a Data Fitting-based application. After you complete development you can disable parameter checking in order to improve the performance of your application.

If you modify the parameter describing dimensions of the arrays that hold the vector-valued function or spline coefficients in contiguous memory, you should call the `df?EditPtr` task editor with the corresponding pointers to the vector-valued function or spline coefficients even when this pointer remains unchanged. Call the `df?EditIdxPtr` editor if those arrays are stored in non-contiguous memory locations.

You must call the `dfiEditVal` task editor to edit the structure of the partition `DF_XHINT` every time you modify a partition using `df?EditPtr`, even if the structure of the partition remains the same.

**df?EditIdxPtr**

*Modifies a pointer to the memory representing a coordinate of the data stored in matrix format.*

**Syntax**

```c
status = dfsEditIdxPtr(task, ptr_attr, idx, ptr)
status = dfdEditIdxPtr(task, ptr_attr, idx, ptr)
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
</tbody>
</table>
**Name** | **Type** | **Description**
---|---|---
ptr_attr  | const MKL_INT  | Type of the data to be modified. The parameter takes one of the values described in "Data Attributes Supported by the df?EditIdxPtr Task Editor".
idx  | const MKL_INT  | Index of the coordinate whose pointer is to be modified.
ptr  | const float* for dfsEditIdxPtr, const double* for dfdEditIdxPtr  | Pointer to the data that holds values of coordinate idx. For details, see table "Data Attributes Supported by the df?EditIdxPtr Task Editor".

**Output Parameters**

**Name** | **Type** | **Description**
---|---|---
status  | int  | Status of the routine:

- DF_STATUS_OK if the routine execution completed successfully.
- Non-zero error code otherwise. See "Task Status and Error Reporting" for error code definitions.

**Description**

The routine modifies a pointer to the array that holds the idx coordinate of vector-valued function y or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the editor if you need to pass into a Data Fitting task or modify the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations. Do not use the editor for coordinates at contiguous memory locations in row-major format.

Before calling this editor, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific df?EditPPSpline1D editor.

**Data Attributes Supported by the df?EditIdxPtr Task Editor**

<table>
<thead>
<tr>
<th>Data Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_Y</td>
<td>Vector-valued function y</td>
</tr>
<tr>
<td>DF_PP_SCOEFF</td>
<td>Piecewise polynomial spline coefficients</td>
</tr>
</tbody>
</table>

When using df?EditIdxPtr, you might receive an error code in the following cases:

- You passed an unsupported parameter value into the editor.
- The value of the index exceeds the predefined value that equals the dimension ny of the vector-valued function.
- You pass a NULL pointer to the editor. In this case, the task remains unchanged.
- You pass a pointer to the idx coordinate of the vector-valued function you provided to contiguous memory in column-major format.

The code example below demonstrates how to use the editor for providing values of a vector-valued function stored in two non-contiguous arrays:

```c
#define NX 1000  /* number of break points */
#define NY 2  /* dimension of vector-valued function */
```
int main()
{
    DFTaskPtr task;
    double x[NX];
    double y1[NX], y2[NX]; /* vector-valued function is stored as two arrays */
    /* Provide first coordinate of two-dimensional function y into creation routine */
    status = dfdNewTask1D( &task, NX, x, DF_NON_UNIFORM_PARTITION, NY, y1,
                           DF_1ST_COORDINATE );
    /* Provide second coordinate of two–dimensional function */
    status = dfdEditIdxPtr( task, DF_Y, 1, y2 );
    ...
}

df?QueryPtr
Reads a pointer to an array held in a Data Fitting task descriptor.

Syntax

status = dfsQueryPtr(task, ptr_attr, ptr)
status = dfdQueryPtr(task, ptr_attr, ptr)

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>ptr_attr</td>
<td>const MKL_INT</td>
<td>The parameter to query. The query routine supports pointer attributes described in the table &quot;Pointers Supported by the df?EditPtr Task Editor&quot;. For details, see the Pointer Attribute column in the table.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>float** for dfsQueryPtr</td>
<td>Pointer to array returned by the query routine. For details, see the Purpose column in table &quot;Pointers Supported by the df?EditPtr Task Editor&quot;.</td>
</tr>
<tr>
<td></td>
<td>double** for dfdQueryPtr</td>
<td></td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code otherwise. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

Description

The df?QueryPtr routine returns the pointer of type ptr_attr stored in a Data Fitting task descriptor as parameter ptr. Attributes of the pointers supported by the query function are identical to those supported by the editor df?EditPtr editor in the table "Pointers Supported by the df?EditPtr Task Editor".
You can use `df?QueryPtr` to read different types of pointers including pointers to the vector-valued function and spline coefficients stored in contiguous memory.

**dfiQueryVal**

*Reads a parameter value in a Data Fitting task descriptor.*

**Syntax**

```c
status = dfiQueryVal(task, val_attr, val)
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task</code></td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td><code>val_attr</code></td>
<td>const MKL_INT</td>
<td>The parameter to query. The query function supports the parameter attributes described in &quot;Parameters Supported by the dfiEditVal Task Editor&quot;.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val</code></td>
<td>MKL_INT</td>
<td>The parameter value returned by the query function. See table &quot;Parameters Supported by the dfiEditVal Task Editor&quot;.</td>
</tr>
<tr>
<td><code>status</code></td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <code>DF_STATUS_OK</code> if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code otherwise. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

**Description**

The `dfiQueryVal` routine returns a parameter of type `val_attr` stored in a Data Fitting task descriptor as parameter `val`. The query function supports the parameter attributes described in "Parameters Supported by the dfiEditVal Task Editor".

**dfpQueryIdxPtr**

*Reads a pointer to the memory representing a coordinate of the data stored in matrix format.*

**Syntax**

```c
status = dfsQueryIdxPtr(task, ptr_attr, idx, ptr)
status = dfdQueryIdxPtr(task, ptr_attr, idx, ptr)
```
Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>ptr_attr</td>
<td>const MKL_INT</td>
<td>Pointer attribute to query. The parameter takes one of the attributes described in &quot;Data Attributes Supported by the df?EditIdxPtr Task Editor&quot;.</td>
</tr>
<tr>
<td>idx</td>
<td>const MKL_INT</td>
<td>Index of the coordinate of the pointer to query.</td>
</tr>
</tbody>
</table>

Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ptr</td>
<td>float* for dfsQueryIdxPtr, double* for dfdQueryIdxPtr</td>
<td>Pointer to the data that holds values of coordinate idx returned. For details, see table &quot;Data Attributes Supported by the df?EditIdxPtr Task Editor&quot;.</td>
</tr>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code otherwise. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

Description

The routine returns a pointer to the array that holds the idx coordinate of vector-valued function y or the pointer to the array of spline coefficients corresponding to the given coordinate.

You can use the query routine if you need the pointer to coordinates of the vector-valued function or spline coefficients held at non-contiguous memory locations or at a contiguous memory location in row-major format (the default storage format for spline coefficients).

Before calling this query routine, make sure that you have created and initialized the task using a task creation function or a relevant editor such as the generic or specific df?EditPPSpline1D editor.

When using df?QueryIdxPtr, you might receive an error code in the following cases:

• You passed an unsupported parameter value into the editor.
• The value of the index exceeds the predefined value that equals the dimension ny of the vector-valued function.
• You request the pointer to the idx coordinate of the vector-valued function you provided to contiguous memory in column-major format.

Data Fitting Computational Routines

Data Fitting computational routines are functions used to perform spline-based computations, such as:

• spline construction
• computation of values, derivatives, and integrals of the predefined order
• cell search
Once you create a Data Fitting task and initialize the required parameters, you can call computational routines as many times as necessary.

The table below lists the available computational routines:

**Data Fitting Computational Routines**

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>df?Construct1D</td>
<td>Constructs a spline for a one-dimensional Data Fitting task.</td>
</tr>
<tr>
<td>df?Interpolate1D</td>
<td>Computes spline values and derivatives.</td>
</tr>
<tr>
<td>df?InterpolateEx1D</td>
<td>Computes spline values and derivatives by calling user-provided interpolants.</td>
</tr>
<tr>
<td>df?Integrate1D</td>
<td>Computes spline-based integrals.</td>
</tr>
<tr>
<td>df?IntegrateEx1D</td>
<td>Computes spline-based integrals by calling user-provided integrators.</td>
</tr>
<tr>
<td>df?SearchCells1D</td>
<td>Finds indices of cells containing interpolation sites.</td>
</tr>
<tr>
<td>df?SearchCellsEx1D</td>
<td>Finds indices of cells containing interpolation sites by calling user-provided cell searchers.</td>
</tr>
</tbody>
</table>

If a Data Fitting computation completes successfully, the computational routines return the `DF_STATUS_OK` code. If an error occurs, the routines return an error code specifying the origin of the failure. Some possible errors are the following:

- The task pointer is `NULL`.
- Memory allocation failed.
- The computation failed for another reason.

For the list of available status codes, see "Task Status and Error Reporting".

**NOTE**

Data Fitting computational routines do not control errors for floating-point conditions, such as overflow, gradual underflow, or operations with Not a Number (NaN) values.

---

**df?Construct1D**

**Syntax**

Constructs a spline of the given type.

```c
status = dfsConstruct1D(task, s_format, method)
status = dfdConstruct1D(task, s_format, method)
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
</tbody>
</table>
### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s_format</td>
<td>const MKL_INT</td>
<td>Spline format. The supported value is DF_PP_SPLINE.</td>
</tr>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Construction method. The supported value is DF_METHOD_STD.</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code if the routine execution failed. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

### Description

Before calling `df?Construct1D`, you need to create and initialize the task, and set the parameters representing the spline. Then you can call the `df?Construct1D` routine to construct the spline. The format of the spline is defined by parameter `s_format`. The method for spline construction is defined by parameter `method`. Upon successful construction, the spline coefficients are available in the user-provided memory location in the format you set through the Data Fitting editor. For the available storage formats, see table "Hint Values for Spline Coefficients".

### Syntax

**df?Interpolate1D/df?InterpolateEx1D**  
Runs data fitting computations.

```c
status = dfsInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell)  
status = dfdInterpolate1D(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell)  
status = dfsInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params)  
status = dfdInterpolateEx1D(task, type, method, nsite, site, sitehint, ndorder, dorder, datahint, r, rhint, cell, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params)
```

### Include Files

- **mkl.h**

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>type</td>
<td>const MKL_INT</td>
<td>Type of spline-based computations. The parameter takes one or more values combined with an \text{OR} \text{operation}. For the list of possible values, see table &quot;Computation Types Supported by the dfsInterpolate1D/ dfsInterpolateEx1D Routines&quot;.</td>
</tr>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Computation method. The supported value is DF_METHOD_PP.</td>
</tr>
<tr>
<td>nsite</td>
<td>const MKL_INT</td>
<td>Number of interpolation sites.</td>
</tr>
<tr>
<td>site</td>
<td>const float* for dfsInterpolate1D/dfsInterpolateEx1D const double* for dfdInterpolate1D/dfdInterpolateEx1D</td>
<td>Array of interpolation sites of size \text{nsite}. The structure of the array is defined by the \text{sitehint} parameter: \begin{itemize} \item If sites form a non-uniform partition, the array should contain \text{nsite} values. \item If sites form a uniform partition, the array should contain two entries that represent the left and the right interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point. \end{itemize}</td>
</tr>
<tr>
<td>sitehint</td>
<td>const MKL_INT</td>
<td>A flag describing the structure of the interpolation sites. For the list of possible values of \text{sitehint}, see table &quot;Hint Values for Interpolation Sites&quot;. If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.</td>
</tr>
<tr>
<td>ndorder</td>
<td>const MKL_INT</td>
<td>Maximal derivative order increased by one to be computed at interpolation sites.</td>
</tr>
<tr>
<td>dorder</td>
<td>const MKL_INT*</td>
<td>Array of size \text{ndorder} that defines the order of the derivatives to be computed at the interpolation sites. If all the elements in \text{dorder} are zero, the library computes the spline values only. If you do not need interpolation computations, set \text{ndorder} to zero and pass a NULL pointer to \text{dorder}.</td>
</tr>
<tr>
<td>datahint</td>
<td>const float* for dfsInterpolate1D/dfsInterpolateEx1D const double* for dfdInterpolate1D/dfdInterpolateEx1D</td>
<td>Array that contains additional information about the structure of partition \text{x} and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the \text{datahint} array, see table &quot;Structure of the datahint Array&quot;.</td>
</tr>
<tr>
<td>r</td>
<td>float* for dfsInterpolate1D/dfsInterpolateEx1D double* for dfdInterpolate1D/dfdInterpolateEx1D</td>
<td>Array for results. If you do not need spline-based interpolation, set this pointer to NULL.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>rhint</td>
<td>const MKL_INT</td>
<td>A flag describing the structure of the results. For the list of possible values of rhint, see table &quot;Hint Values for the rhint Parameter&quot;. If you set the flag to DF_NO_HINT, the library stores the result in row-major format.</td>
</tr>
<tr>
<td>cell</td>
<td>MKL_INT*</td>
<td>Array of cell indices in partition ( x ) that contain the interpolation sites. Provide this parameter as input if type is DF_INTERP_USER_CELL. If you do not need cell indices, set this parameter to NULL.</td>
</tr>
<tr>
<td>le_cb</td>
<td>const dfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D</td>
<td>User-defined callback function for extrapolation at the sites to the left of the interpolation interval. Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>le_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the library to the le_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>re_cb</td>
<td>const dfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D</td>
<td>User-defined callback function for extrapolation at the sites to the right of the interpolation interval. Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>re_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the library to the re_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>i_cb</td>
<td>const dfsInterpCallBack for dfsInterpolateEx1D constdfdInterpCallBack for dfdInterpolateEx1D</td>
<td>User-defined callback function for interpolation within the interpolation interval. Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>i_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the library to the i_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>search_cb</td>
<td>const dfsSearchCellsCallBack for dfsInterpolateEx1D constdfdSearchCellsCallBack for dfdInterpolateEx1D</td>
<td>User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>search_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the library to the search_cb function.</td>
</tr>
</tbody>
</table>
Set to NULL if there are no additional parameters or if you are not supplying a callback function.

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code if the routine execution failed. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
<tr>
<td>r</td>
<td></td>
<td>Contains results of computations at the interpolation sites.</td>
</tr>
<tr>
<td>cell</td>
<td></td>
<td>Array of cell indices in partition x that contain the interpolation sites, which is computed if type is DF_CELL.</td>
</tr>
</tbody>
</table>

### Description

The df?Interpolate1D/df?InterpolateEx1D routine performs spline-based computations with user-defined settings. The routine supports two types of computations for interpolation sites provided in array site:

#### Computation Types Supported by the df?Interpolate1D/df?InterpolateEx1D Routines

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_INTERP</td>
<td>Compute derivatives of predefined order. The derivative of the zero order is the spline value.</td>
</tr>
<tr>
<td>DF_INTERP_USER_CELL</td>
<td>Compute derivatives of predefined order given user-provided cell indices. The derivative of the zero order is the spline value. For this type of the computations you should provide a valid cell array, which holds the indices of cells in the site array containing relevant interpolation sites.</td>
</tr>
<tr>
<td>DF_CELL</td>
<td>Compute indices of cells in partition x that contain the sites.</td>
</tr>
</tbody>
</table>

If the indices of cells which contain interpolation types are available before the call to df?Interpolate1D/df?InterpolateEx1D, you can improve performance by using the DF_INTERP_USER_CELL computation type.
NOTE
If you pass any combination of DF_INTERP, DF_INTERP_USER_CELL, and DF_CELL computation types to the routine, the library uses the DF_INTERP_USER_CELL computation mode.

If you specify DF_INTERP_USER_CELL computation mode and a user-defined callback function for computing cell indices to df?InterpolateEx1D, the library uses the DF_INTERP_USER_CELL computation mode, and the call-back function is not called.

If the sites do not belong to interpolation interval \([a, b]\), the library uses:
- polynomial \(P_0\) of the spline constructed on interval \([x_0, x_1]\) for computations at the sites to the left of \(a\).
- polynomial \(P_{n-2}\) of the spline constructed on interval \([x_{n-2}, x_{n-1}]\) for computations at the sites to the right of \(b\).

Interpolation sites support the following hints:

**Hint Values for Interpolation Sites**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_NON_UNIFORM_PARTITION</td>
<td>Partition is non-uniform.</td>
</tr>
<tr>
<td>DF_UNIFORM_PARTITION</td>
<td>Partition is uniform.</td>
</tr>
<tr>
<td>DF_SORTED_DATA</td>
<td>Interpolation sites are sorted in the ascending order and define a non-uniform partition.</td>
</tr>
<tr>
<td>DF_NO_HINT</td>
<td>No hint is provided. By default, the partition defined by interpolation sites is interpreted as non-uniform.</td>
</tr>
</tbody>
</table>

NOTE
If you pass a sorted array of interpolation sites to the Intel® MKL, set the sitehint parameter to the DF_SORTED_DATA value. The library uses this information when choosing the search algorithm and ignores any other data hints about the structure of the interpolation sites.

Data Fitting computation routines can use the following hints to speed up the computation:
- DF_UNIFORM_PARTITION describes the structure of breakpoints and the interpolation sites.
- DF_QUASI_UNIFORM_PARTITION describes the structure of breakpoints.

Pass the above hints to the library when appropriate.

For spline-based interpolation, you should set the derivatives whose values are required for the computation. You can provide the derivatives by setting the dorder array of size ndorder as follows:

\[
dorder[i] = \begin{cases} 
1, & \text{if derivative of the } i\text{-th order is required} \\
0, & \text{otherwise} 
\end{cases} \quad i = 0, ..., ndorder - 1
\]

Orders of derivatives \(i_d(d = 0, 1, .., nder - 1)\), corresponding to non-zero derivatives to be calculated, form the array \(\{i_d\}\) of length \(nder \leq ndorder\).

The storage format for the interpolation results is specified using the rhint parameter values. For each storage format, Table Hint Values for the rhint Parameter describes how to get the result \(R(j, s, i_d)\) from array \(r\), for function index \(j(0 \leq jn - 1)\), site number \(s(0 \leq nsite - 1)\), and derivative index \(i_d(0 \leq nnder - 1)\), where \(jn\) is the number of functions, \(nsite\) is the number of sites, and \(nder\) is the total number of non-zero derivatives for interpolation. The array \(r\) can be either a one-dimensional array of size \(ny \times nder \times nsite\) or a three-dimensional array with the dimensions described in the table.
### Hint Values for the `rhint` Parameter

<table>
<thead>
<tr>
<th>Value</th>
<th>Location of $R(j, s, i_d)$, One-dimensional Array Storage</th>
<th>Location of $R(j, s, i_d)$, Three-dimensional Array Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_MATRIX_STORAGE_FUNCS_SITES_DERS</td>
<td>$r[d + nder*(s + nsite*j)]$</td>
<td>$r[j][s][d]$</td>
</tr>
<tr>
<td>DF_MATRIX_STORAGE_FUNCS_DERS_SITES</td>
<td>$r[s + nsite*(d + nder*j)]$</td>
<td>$r[j][d][s]$</td>
</tr>
<tr>
<td>DF_MATRIX_STORAGE_SITES_FUNCS_DERS</td>
<td>$r[d + nder*(j + ny*s)]$</td>
<td>$r[s][j][d]$</td>
</tr>
<tr>
<td>DF_MATRIX_STORAGE_SITES_DERS_FUNCS</td>
<td>$r[j + ny*(d + nder*s)]$</td>
<td>$r[s][d][j]$</td>
</tr>
<tr>
<td>DF_NO_HINT</td>
<td>No hint is provided. By default, the results are stored as in <code>rhint = DF_MATRIX_STORAGE_FUNCS_SITES_DERS</code>.</td>
<td></td>
</tr>
</tbody>
</table>

The following figures show the structure of the storage formats. Each shows sequential memory layout line by line, left to right.

- **Storage in $r$ for `rhint = DF_MATRIX_STORAGE_FUNCS_SITES_DERS (DF_MATRIX_STORAGE_ROWS)`:**

<table>
<thead>
<tr>
<th>$R(0,0,i_0)$</th>
<th>$R(0,0,i_1)$</th>
<th>...</th>
<th>$R(0,0,i_{nder-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(0,1,i_0)$</td>
<td>$R(0,1,i_1)$</td>
<td>...</td>
<td>$R(0,1,i_{nder-1})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$R(0,nsite-1,i_0)$</td>
<td>$R(0,nsite-1,i_1)$</td>
<td>...</td>
<td>$R(0,nsite-1,i_{nder-1})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R(1,0,i_0)$</th>
<th>$R(1,0,i_1)$</th>
<th>...</th>
<th>$R(1,0,i_{nder-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(1,1,i_0)$</td>
<td>$R(1,1,i_1)$</td>
<td>...</td>
<td>$R(1,1,i_{nder-1})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$R(1,nsite-1,i_0)$</td>
<td>$R(1,nsite-1,i_1)$</td>
<td>...</td>
<td>$R(1,nsite-1,i_{nder-1})$</td>
</tr>
</tbody>
</table>

- **Storage in $r$ for `rhint = DF_MATRIX_STORAGE_FUNCS_DERS_SITES (DF_MATRIX_STORAGE_COLS)`:**

<table>
<thead>
<tr>
<th>$R(0,0,i_0)$</th>
<th>$R(1,0,i_1)$</th>
<th>...</th>
<th>$R(0,0,i_{nder-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(0,1,i_0)$</td>
<td>$R(1,1,i_1)$</td>
<td>...</td>
<td>$R(0,1,i_{nder-1})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$R(0,nsite-1,i_0)$</td>
<td>$R(1,nsite-1,i_1)$</td>
<td>...</td>
<td>$R(0,nsite-1,i_{nder-1})$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$R(1,0,i_0)$</th>
<th>$R(1,0,i_1)$</th>
<th>...</th>
<th>$R(1,0,i_{nder-1})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R(1,1,i_0)$</td>
<td>$R(1,1,i_1)$</td>
<td>...</td>
<td>$R(1,1,i_{nder-1})$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$R(1,nsite-1,i_0)$</td>
<td>$R(1,nsite-1,i_1)$</td>
<td>...</td>
<td>$R(1,nsite-1,i_{nder-1})$</td>
</tr>
</tbody>
</table>

...
\begin{align*}
R(0, 0, i_0) & \quad R(0, 1, i_0) & \quad \ldots & \quad R(0, nsite - 1, i_0) \\
R(0, 0, i_1) & \quad R(0, 1, i_1) & \quad \ldots & \quad R(0, nsite - 1, i_1) \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
R(0, 0, i_{nder - 1}) & \quad R(0, 1, i_{nder - 1}) & \quad \ldots & \quad R(0, nsite - 1, i_{nder - 1}) \\
\end{align*}

\begin{align*}
R(1, 0, i_0) & \quad R(1, 1, i_0) & \quad \ldots & \quad R(1, nsite - 1, i_0) \\
R(1, 0, i_1) & \quad R(1, 1, i_1) & \quad \ldots & \quad R(1, nsite - 1, i_1) \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
R(1, 0, i_{nder - 1}) & \quad R(1, 1, i_{nder - 1}) & \quad \ldots & \quad R(1, nsite - 1, i_{nder - 1}) \\
\end{align*}

\begin{align*}
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
\end{align*}

- **Storage in** \( x \) **for** \( rhint = \text{DF\_MATRIX\_STORAGE\_SITES\_FUNCS\_DERS} \):

\begin{align*}
R(0, 0, i_0) & \quad R(0, 0, i_1) & \quad \ldots & \quad R(0, 0, i_{nder - 1}) \\
R(1, 0, i_0) & \quad R(1, 0, i_1) & \quad \ldots & \quad R(1, 0, i_{nder - 1}) \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
R(ny - 1, 0, i_0) & \quad R(ny - 1, 0, i_1) & \quad \ldots & \quad R(ny - 1, 0, i_{nder - 1}) \\
\end{align*}

\begin{align*}
R(0, 1, i_0) & \quad R(0, 1, i_1) & \quad \ldots & \quad R(0, 1, i_{nder - 1}) \\
R(1, 1, i_0) & \quad R(1, 1, i_1) & \quad \ldots & \quad R(1, 1, i_{nder - 1}) \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
R(ny - 1, 1, i_0) & \quad R(ny - 1, 1, i_1) & \quad \ldots & \quad R(ny - 1, 1, i_{nder - 1}) \\
\end{align*}

\begin{align*}
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
\end{align*}

- **Storage in** \( x \) **for** \( rhint = \text{DF\_MATRIX\_STORAGE\_SITES\_DERS\_FUNCS} \):

\begin{align*}
R(0, 0, i_0) & \quad R(1, 0, i_0) & \quad \ldots & \quad R(ny - 1, 0, i_0) \\
R(0, 0, i_1) & \quad R(1, 0, i_1) & \quad \ldots & \quad R(ny - 1, 0, i_1) \\
\ldots & \quad \ldots & \quad \ldots & \quad \ldots \\
R(0, 0, i_{nder - 1}) & \quad R(1, 0, i_{nder - 1}) & \quad \ldots & \quad R(ny - 1, 0, i_{nder - 1}) \\
\end{align*}
To speed up Data Fitting computations, use the `datahint` parameter that provides additional information about the structure of the partition and interpolation sites. This data represents a floating-point or a double array with the following structure:

### Structure of the `datahint` Array

<table>
<thead>
<tr>
<th>Element Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Task dimension</td>
</tr>
<tr>
<td>1</td>
<td>Type of additional information</td>
</tr>
<tr>
<td>2</td>
<td>Reserved field</td>
</tr>
<tr>
<td>3</td>
<td>The total number ( q ) of elements containing additional information.</td>
</tr>
<tr>
<td>4</td>
<td>Element (1)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( q+3 )</td>
<td>Element (( q ))</td>
</tr>
</tbody>
</table>

Data Fitting computation functions support the following types of additional information for `datahint[1]`:

### Types of Additional Information

<table>
<thead>
<tr>
<th>Type</th>
<th>Element Number</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DF_NO_APRIORI_INFO</code></td>
<td>0</td>
<td>No parameters are provided. Information about the data structure is absent.</td>
</tr>
<tr>
<td><code>DF_APRIORI_MOST_LIKELY_CELL</code></td>
<td>1</td>
<td>Index of the cell that is likely to contain interpolation sites.</td>
</tr>
</tbody>
</table>

To compute indices of the cells that contain interpolation sites, provide the pointer to the array of size \( nsite \) for the results. The library supports the following scheme of cell indexing for the given partition \( \{ x_i \} \), \( i=1,\ldots,nx \):

\[
\begin{align*}
\text{cell}[j] &= i, \text{if} \ \text{site}[j] \in [x_i, x_{i+1}), \ i = 0,\ldots, nx - 2, \\
\text{cell}[j] &= nx - 1, \text{if} \ \text{site}[j] \in [x_{nx - 1}, x_{nx}], \\
\text{cell}[j] &= nx, \text{if} \ \text{site}[j] \in (x_{nx}, x_{nx + 1}],
\end{align*}
\]

where
- \( x_0 = -\infty \)
- \( x_{nx+1} = +\infty \)
To perform interpolation computations with spline types unsupported in the Data Fitting component, use the extended version of the routine `df?InterpolateEx1D`. With this routine, you can provide user-defined callback functions for computations within, to the left of, or to the right of interpolation interval \([a, b]\). The callback functions compute indices of the cells that contain the specified interpolation sites or can serve as an approximation for computing the exact indices of such cells.

If you do not pass any function for computations at the sites outside the interval \([a, b]\), the routine uses the default settings.

**See Also**

Mathematical Conventions for Data Fitting Functions

df?InterpCallBack
df?SearchCellsCallBack

**df?Integrate1D/** df?IntegrateEx1D

*Computes a spline-based integral.*

**Syntax**

```c
status = dfsIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint, rdatahint, \( r, rhint \))
status = dfdIntegrate1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint, rdatahint, \( r, rhint \))
status = dfsIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint, rdatahint, \( r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params \))
status = dfdIntegrateEx1D(task, method, nlim, llim, llimhint, rlim, rlimhint, ldatahint, rdatahint, \( r, rhint, le_cb, le_params, re_cb, re_params, i_cb, i_params, search_cb, search_params \))
```

**Include Files**

- mkl.h

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Integration method. The supported value is <code>DF_METHOD_PP</code>.</td>
</tr>
<tr>
<td>nlim</td>
<td>const MKL_INT</td>
<td>Number of pairs of integration limits.</td>
</tr>
<tr>
<td>llim</td>
<td>const float* for dfsIntegrate1D/dfsIntegrateEx1D</td>
<td>Array of size <code>nlim</code> that defines the left-side integration limits.</td>
</tr>
<tr>
<td></td>
<td>const double* for dfdIntegrate1D/dfdIntegrateEx1D</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>llimhint</td>
<td>const MKL_INT</td>
<td>A flag describing the structure of the left-side integration limits llim. For the list of possible values of llimhint, see table &quot;Hint Values for Integration Limits&quot;. If you set the flag to the DF_NO_HINT value, the library assumes that the left-side integration limits define a non-uniform partition.</td>
</tr>
<tr>
<td>rlim</td>
<td>const float* for dfsIntegrate1D/dfsIntegrateEx1D, const double* for dfdIntegrate1D/dfdIntegrateEx1D</td>
<td>Array of size nlim that defines the right-side integration limits.</td>
</tr>
<tr>
<td>rlimhint</td>
<td>const MKL_INT</td>
<td>A flag describing the structure of the right-side integration limits rlim. For the list of possible values of rlimhint, see table &quot;Hint Values for Integration Limits&quot;. If you set the flag to the DF_NO_HINT value, the library assumes that the right-side integration limits define a non-uniform partition.</td>
</tr>
<tr>
<td>idatahint</td>
<td>const float* for dfsIntegrate1D/dfsIntegrateEx1D, const double* for dfdIntegrate1D/dfdIntegrateEx1D</td>
<td>Array that contains additional information about the structure of partition x and left-side integration limits. For details on the idatahint array, see table &quot;Structure of the datahint Array&quot; in the description of the df? Interpolate1D function.</td>
</tr>
<tr>
<td>rdatahint</td>
<td>const float* for dfsIntegrate1D/dfsIntegrateEx1D, const double* for dfdIntegrate1D/dfdIntegrateEx1D</td>
<td>Array that contains additional information about the structure of partition x and right-side integration limits. For details on the rdatahint array, see table &quot;Structure of the datahint Array&quot; in the description of the df? Interpolate1D function.</td>
</tr>
<tr>
<td>rhint</td>
<td>const MKL_INT</td>
<td>A flag describing the structure of the results. For the list of possible values of rhint, see table &quot;Hint Values for Integration Results&quot;. If you set the flag to the DF_NO_HINT value, the library stores the results in row-major format.</td>
</tr>
<tr>
<td>le_cb</td>
<td>const dfsIntegrCallBack for dfsIntegrateEx1D, const dfdIntegrCallBack for dfdIntegrateEx1D</td>
<td>User-defined callback function for integration on interval ([llim[i], \min(rlim[i], a))) for (llim[i] &lt; a). Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>le_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the library to the le_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>re_cb</td>
<td>const dfsInterpCallBack for dfsIntegrateEx1D</td>
<td>User-defined callback function for integration on interval [\max(\text{llim}[i], b), rlim[i]) for (rlim[i] \geq b). Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>re_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the re_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>i_cb</td>
<td>const dfsIntegrCallBack for dfsIntegrateEx1D</td>
<td>User-defined callback function for integration on interval [\max(a, \text{llim}[i]), \min(rlim[i], b)). Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>i_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the i_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
<tr>
<td>search_cb</td>
<td>const dfsSearchCellsCalBack for dfsIntegrateEx1D</td>
<td>User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.</td>
</tr>
<tr>
<td>search_params</td>
<td>const void*</td>
<td>Pointer to additional user-defined parameters passed by the search_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DF_STATUS_OK if the routine execution completed successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code if the routine execution failed. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
<tr>
<td>r</td>
<td>float* for dfsIntegrate1D/dfsIntegrateEx1D</td>
<td>Array of integration results. The size of the array should be sufficient to hold (nlim<em>ny) values, where (ny) is the dimension of the vector-valued function. The integration results are packed according to the settings in rhint. double</em> for dfdIntegrate1D/dfdIntegrateEx1D</td>
</tr>
</tbody>
</table>
Caution
The \texttt{df?Integrate1D/df?IntegrateEx1D} routines do not support in-place computations. You must provide non-aliasing memory locations for the \texttt{llim} and \texttt{rlim} input parameters and the \texttt{r} output parameter.

Description
The \texttt{df?Integrate1D/df?IntegrateEx1D} routine computes spline-based integral on user-defined intervals

$$I(i,j) = \int_{r_i}^{r_j} f_j(x) \, dx,$$

where \( r_i = rlim[i] \), \( l_i = llim[i] \), and \( i = 0, \ldots, ny - 1 \).

If \( rlim[i] < llim[i] \), the routine returns

$$I(i,j) = -\int_{l_i}^{r_i} f_j(x) \, dx$$

The routine supports the following hint values for integration results:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{DF_MATRIX_STORAGE_ROWS}</td>
<td>Data is stored in row-major format according to C conventions.</td>
</tr>
<tr>
<td>\texttt{DF_MATRIX_STORAGE_COLS}</td>
<td>Data is stored in column-major format according to Fortran conventions.</td>
</tr>
<tr>
<td>\texttt{DF_NO_HINT}</td>
<td>No hint is provided. By default, the coordinates of vector-valued function ( y ) are provided and stored in row-major format.</td>
</tr>
</tbody>
</table>

A common structure of the storage formats for the integration results is as follows:

**Row-major format**

\[
\begin{array}{cccc}
I(0, 0) & \ldots & \ldots & I(0, nlim - 1) \\
\ldots & \ldots & \ldots & \ldots \\
I(ny - 1, 0) & \ldots & \ldots & I(ny - 1, nlim - 1)
\end{array}
\]

**Column-major format**

\[
\begin{array}{cccc}
I(0, 0) & \ldots & \ldots & I(ny - 1, 0) \\
\ldots & \ldots & \ldots & \ldots \\
I(0, nlim - 1) & \ldots & \ldots & I(ny - 1, nlim - 1)
\end{array}
\]

Using the \texttt{llimhint} and \texttt{rlimhint} parameters, you can provide the following hint values for integration limits:
### Hint Values for Integration Limits

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DF_SORTED_DATA</td>
<td>Integration limits are sorted in the ascending order and define a non-uniform partition.</td>
</tr>
<tr>
<td>DF_NON_UNIFORM_PARTITION</td>
<td>Partition defined by integration limits is non-uniform.</td>
</tr>
<tr>
<td>DF_UNIFORM_PARTITION</td>
<td>Partition defined by integration limits is uniform.</td>
</tr>
<tr>
<td>DF_NO_HINT</td>
<td>No hint is provided. By default, partition defined by integration limits is interpreted as non-uniform.</td>
</tr>
</tbody>
</table>

To compute integration with splines unsupported in the Data Fitting component, use the extended version of the routine `df?IntegrateEx1D`. With this routine, you can provide user-defined callback functions that compute:

- integrals within, to the left of, or to the right of the interpolation interval \([a, b]\)
- indices of cells that contain the provided integration limits or can serve as an approximation for computing the exact indices of such cells

If you do not pass callback functions, the routine uses the default settings.

### See Also

Mathematical Conventions for Data Fitting Functions

- `df?Interpolate1D/df?InterpolateEx1D`
- `df?IntegrCallBack`
- `df?SearchCellsCallBack`
- `df?SearchCells1D/df?SearchCellsEx1D`

**Searches sub-intervals containing interpolation sites.**

### Syntax

```c
status = dfsSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
status = dfdSearchCells1D(task, method, nsite, site, sitehint, datahint, cell)
status = dfsSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell, search_cb, search_params)
status = dfdSearchCellsEx1D(task, method, nsite, site, sitehint, datahint, cell, search_cb, search_params)
```

### Include Files

- `mkl.h`

### Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task.</td>
</tr>
<tr>
<td>method</td>
<td>const MKL_INT</td>
<td>Search method. The supported value is DF_METHOD_STD.</td>
</tr>
</tbody>
</table>
### Name | Type | Description
--- | --- | ---
nsite | const MKL_INT* | Number of interpolation sites.
site | const float* for dfsSearchCells1D/ dfsSearchCellsEx1D, const double* for dfdSearchCells1D/ dfdSearchCellsEx1D | Array of interpolation sites of size nsite. The structure of the array is defined by the sitehint parameter:
- If the sites form a non-uniform partition, the array should contain nsite values.
- If the sites form a uniform partition, the array should contain two entries that represent the left-most and the right-most interpolation sites. The first entry of the array contains the left-most interpolation point. The second entry of the array contains the right-most interpolation point.
sitehint | const MKL_INT | A flag describing the structure of the interpolation sites. For the list of possible values of sitehint, see table "Hint Values for Interpolation Sites". If you set the flag to DF_NO_HINT, the library interprets the site-defined partition as non-uniform.
datahint | const float* for dfsSearchCells1D/ dfsSearchCellsEx1D, const double* for dfdSearchCells1D/ dfdSearchCellsEx1D | Array that contains additional information about the structure of the partition and interpolation sites. This data helps to speed up the computation. If you provide a NULL pointer, the routine uses the default settings for computations. For details on the datahint array, see table "Structure of the datahint Array".
search_cb | const dfsSearchCellsCallBac k for dfsSearchCellsEx1D, const dfdSearchCellsCallBac k for dfdSearchCellsEx1D | User-defined callback function for computing indices of cells that can contain interpolation sites. Set to NULL if you are not supplying a callback function.
search_params | const void* | Pointer to additional user-defined parameters passed by the library to the search_cb function. Set to NULL if there are no additional parameters or if you are not supplying a callback function.

### Output Parameters

### Name | Type | Description
--- | --- | ---
status | int | Status of the routine:
- DF_STATUS_OK if the routine execution completed successfully.
- Non-zero error code if the routine execution failed. See "Task Status and Error Reporting" for error code definitions.
cell | MKL_INT* | Array of cell indices in the partition that contain the interpolation sites.
Description

The df?SearchCells1D/df?SearchCellsEx1D routines return array cell of indices of sub-intervals (cells) in the partition that contain interpolation sites available in array site. For details on the cell indexing scheme, see the description of the df?Interpolate1D/df?InterpolateEx1D computation routines.

Use the datahint parameter to provide additional information about the structure of the partition and/or interpolation sites. The definition of the datahint parameter is availalbe in the description of the df?Interpolate1D/df?InterpolateEx1D computation routines.

For description of the user-defined callback for computation of cell indices, see df?SearchCellsCallBack.

See Also
Mathematical Conventions for Data Fitting Functions
df?Interpolate1D/df?InterpolateEx1D

df?SearchCellsCallBack

df?InterpCallBack

A callback function for user-defined interpolation to be passed into df?InterpolateEx1D.

Syntax

\[
status = \text{dfsInterpCallBack}(n, \text{cell}, \text{site}, r, \text{user_params}, \text{library_params})
\]

\[
status = \text{dfdInterpCallBack}(n, \text{cell}, \text{site}, r, \text{user_params}, \text{library_params})
\]

Include Files

- mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>long long*</td>
<td>Number of interpolation sites.</td>
</tr>
<tr>
<td>cell</td>
<td>long long*</td>
<td>Array of size ( n ) containing indices of the cells to which the interpolation sites in array ( site ) belong.</td>
</tr>
<tr>
<td>site</td>
<td>float* for dfsInterpCallBack double* for dfdInterpCallBack</td>
<td>Array of interpolation sites of size ( n ).</td>
</tr>
<tr>
<td>user_params</td>
<td>void*</td>
<td>Pointer to user-defined parameters of the callback function.</td>
</tr>
<tr>
<td>library_params</td>
<td>dfInterpCallBackLibraryParameters*</td>
<td>Pointer to library-defined parameters of the callback function.</td>
</tr>
</tbody>
</table>
Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>The status returned by the callback function:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Zero indicates successful completion of the callback</td>
</tr>
<tr>
<td></td>
<td></td>
<td>operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A negative value indicates an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A positive value indicates a warning.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See &quot;Task Status and Error Reporting&quot; for error code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>definitions.</td>
</tr>
<tr>
<td>r</td>
<td>float*</td>
<td>for dfsInterpCallBack</td>
</tr>
<tr>
<td></td>
<td>double*</td>
<td>for dfdInterpCallBack</td>
</tr>
</tbody>
</table>

Array of the computed interpolation results packed in row-
major format.

Description

When passed into the df?InterpolateEx1D routine, this function performs user-defined interpolation
operation.

The library_params parameter allows the library to provide extra parameters. Currently no parameters are
provided.

See Also
df?interpolate1d/df?interpolateex1d
df?searchcellscallback

df?IntegrCallBack

A callback function that you can pass into df?
IntegrateEx1D to define integration computations.

Syntax

status = dfsIntegrCallBack(n, lcell, llim, rcell, rlim, r, user_params, library_params)
status = dfdIntegrCallBack(n, lcell, llim, rcell, rlim, r, user_params, library_params)

Include Files

• mkl.h

Input Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>long long*</td>
<td>Number of pairs of integration limits.</td>
</tr>
</tbody>
</table>
| lcell  | long long* | Array of size n with indices of the cells that contain the left-
<p>|        |           | side integration limits in array llim.                       |
| llim   | float*   | for dfsIntegrCallBack                                         |
|        |          | Array of size n that holds the left-side integration limits.  |</p>
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcell</td>
<td>long long*</td>
<td>Array of size ( n ) with indices of the cells that contain the right-side integration limits in array ( rlim ).</td>
</tr>
<tr>
<td>rlim</td>
<td>float* for dfsIntegrCallBack</td>
<td>Array of size ( n ) that holds the right-side integration limits.</td>
</tr>
<tr>
<td>user_params</td>
<td>void*</td>
<td>Pointer to user-defined parameters of the callback function.</td>
</tr>
<tr>
<td>library_params</td>
<td>dfIntegrCallBackLibraryParams*</td>
<td>Pointer to library-defined parameters of the callback function.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>The status returned by the callback function:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Zero indicates successful completion of the callback operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A negative value indicates an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A positive value indicates a warning.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
<tr>
<td>r</td>
<td>float* for dfsIntegrCallBack</td>
<td>Array of integration results. For packing the results in row-major format, follow the instructions described in df?Interpolate1D/df?InterpolateEx1D.</td>
</tr>
<tr>
<td></td>
<td>double* for dfdIntegrCallBack</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

When passed into the df?IntegrateEx1D routine, this function defines integration computations. If at least one of the integration limits is outside the interpolation interval \([a, b]\), the library decomposes the integration into sub-intervals that belong to the extrapolation range to the left of \( a \), the extrapolation range to the right of \( b \), and the interpolation interval \([a, b]\), as follows:

- If the left integration limit is to the left of the interpolation interval \((llim < a)\), the df?IntegrateEx1D routine passes \( llim \) as the left integration limit and \( \min(rlim, a) \) as the right integration limit to the user-defined callback function.
- If the right integration limit is to the right of the interpolation interval \((rlim > b)\), the df?IntegrateEx1D routine passes \( \max(llim, b) \) as the left integration limit and \( rlim \) as the right integration limit to the user-defined callback function.
- If the left and the right integration limits belong to the interpolation interval, the df?IntegrateEx1D routine passes them to the user-defined callback function unchanged.

The value of the integral is the sum of integral values obtained on the sub-intervals.
The `library_params` parameter allows the library to provide extra parameters. Currently no parameters are provided.

**Optimization Notice**

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**See Also**

- `df?Integrate1D/df?IntegrateEx1D`
- `df?IntegrCallBack`
- `df?SearchCellsCallBack`

**df?SearchCellsCallBack**

A callback function for user-defined search to be passed into `df?InterpolateEx1D`, `df?IntegrateEx1ID`, or `df?SearchCellsEx1D`.

**Syntax**

```c
status = dfsSearchCellsCallBack(n, site, cell, flag, user_params, library_params)
status = dfdSearchCellsCallBack(n, site, cell, flag, user_params, library_params)
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td><code>long long*</code></td>
<td>Number of interpolation sites or integration limits.</td>
</tr>
<tr>
<td>site</td>
<td><code>float*</code> for dfsSearchCellsCallBack <code>double*</code> for dfdSearchCellsCallBack</td>
<td>Array, size <code>n</code>, of interpolation sites or integration limits.</td>
</tr>
<tr>
<td>flag</td>
<td><code>int*</code></td>
<td>Array of size <code>n</code>, with values set as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If the cell with index <code>cell[i]</code> contains <code>site[i]</code>, set <code>flag[i]</code> to 1.</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>user_params</td>
<td>void*</td>
<td>Pointer to user-defined parameters of the callback function.</td>
</tr>
<tr>
<td>library_p</td>
<td>dfSearchCallBackLibraryParams*</td>
<td>Pointer to library-defined parameters of the callback function.</td>
</tr>
</tbody>
</table>

### Output Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>The status returned by the callback function:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Zero indicates successful completion of the callback operation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A negative value indicates an error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• The DF_STATUS_EXACT_RESULT status indicates that cell indices returned by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the callback function are exact. In this case, you do not need to initialize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>entries of the flag array.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• A positive value indicates a warning.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
<tr>
<td>cell</td>
<td>long long*</td>
<td>Array of size n that returns indices of the cells computed by the callback</td>
</tr>
<tr>
<td></td>
<td></td>
<td>function.</td>
</tr>
</tbody>
</table>

**Description**

When passed into the df?InterpolateEx1D, df?IntegrateEx1D, or df?SearchCellsEx1D routine, this function performs a user-defined search.

The library_params parameter allows the library to provide extra parameters. The df?InterpolateEx1D, and df?SearchCellsEx1D routines do not provide extra parameters and set library_params to NULL. The df?IntegrateEx1D routines use this parameter to specify which type of integration limits, left or right, are provided for the callback. To do this the library declares the dfSearchCallBackLibraryParams structure. It currently contains one field, limit_type_flag, of type int. The field is set by the library to one of two possible values: DF_INTEGR_SEARCH_CB_LLIM_FLAG if the left integration limits are provided, or DF_INTEGR_SEARCH_CB_RLIM_FLAG if the right integration limits are provided.

**See Also**

df?Interpolate1D/df?InterpolateEx1D

df?InterpCallBack
Data Fitting Task Destructors

Task destructors are routines used to delete task descriptors and deallocate the corresponding memory resources. The Data Fitting task destructor `dfDeleteTask` destroys a Data Fitting task and frees the memory.

`dfDeleteTask`

*Destroys a Data Fitting task object and frees the memory.*

**Syntax**

```c
status = dfDeleteTask(&task)
```

**Include Files**

- `mkl.h`

**Input Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>task</td>
<td>DFTaskPtr</td>
<td>Descriptor of the task to destroy.</td>
</tr>
</tbody>
</table>

**Output Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>int</td>
<td>Status of the routine:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• <strong>DF_STATUS_OK</strong> if the task is deleted successfully.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Non-zero error code if the operation failed. See &quot;Task Status and Error Reporting&quot; for error code definitions.</td>
</tr>
</tbody>
</table>

**Description**

Given a pointer to a task descriptor, this routine deletes the Data Fitting task descriptor and frees the memory allocated for the structure. If the task is deleted successfully, the routine sets the task pointer to **NULL**. Otherwise, the routine returns an error code.
Linear Solvers Basics

Many applications in science and engineering require the solution of a system of linear equations. This problem is usually expressed mathematically by the matrix-vector equation, \( Ax = b \), where \( A \) is an \( m \times n \) matrix, \( x \) is the \( n \) element column vector and \( b \) is the \( m \) element column vector. The matrix \( A \) is usually referred to as the coefficient matrix, and the vectors \( x \) and \( b \) are referred to as the solution vector and the right-hand side, respectively.

Basic concepts related to solving linear systems with sparse matrices are described in Sparse Linear Systems and various storage schemes for sparse matrices are described in Sparse Matrix Storage Formats.

Sparse Linear Systems

In many real-life applications, most of the elements in \( A \) are zero. Such a matrix is referred to as sparse. Conversely, matrices with very few zero elements are called dense. For sparse matrices, computing the solution to the equation \( Ax = b \) can be made much more efficient with respect to both storage and computation time, if the sparsity of the matrix can be exploited. The more an algorithm can exploit the sparsity without sacrificing the correctness, the better the algorithm.

Generally speaking, computer software that finds solutions to systems of linear equations is called a solver. A solver designed to work specifically on sparse systems of equations is called a sparse solver. Solvers are usually classified into two groups - direct and iterative.

Iterative Solvers start with an initial approximation to a solution and attempt to estimate the difference between the approximation and the true result. Based on the difference, an iterative solver calculates a new approximation that is closer to the true result than the initial approximation. This process is repeated until the difference between the approximation and the true result is sufficiently small. The main drawback to iterative solvers is that the rate of convergence depends greatly on the values in the matrix \( A \). Consequently, it is not possible to predict how long it will take for an iterative solver to produce a solution. In fact, for ill-conditioned matrices, the iterative process will not converge to a solution at all. However, for well-conditioned matrices it is possible for iterative solvers to converge to a solution very quickly. Consequently, if an application involves well-conditioned matrices iterative solvers can be very efficient.

Direct Solvers, on the other hand, factor the matrix \( A \) into the product of two triangular matrices and then perform a forward and backward triangular solve.

This approach makes the time required to solve a systems of linear equations relatively predictable, based on the size of the matrix. In fact, for sparse matrices, the solution time can be predicted based on the number of non-zero elements in the array \( A \).

Matrix Fundamentals

A matrix is a rectangular array of either real or complex numbers. A matrix is denoted by a capital letter; its elements are denoted by the same lower case letter with row/column subscripts. Thus, the value of the element in row \( i \) and column \( j \) in matrix \( A \) is denoted by \( a(i, j) \). For example, a 3 by 4 matrix \( A \), is written as follows:

\[
A = \begin{bmatrix}
  a(1, 1) & a(1, 2) & a(1, 3) & a(1, 4) \\
  a(2, 1) & a(2, 2) & a(2, 3) & a(2, 4) \\
  a(3, 1) & a(3, 2) & a(3, 3) & a(3, 4)
\end{bmatrix}
\]

Note that with the above notation, we assume the standard Fortran programming language convention of starting array indices at 1 rather than the C programming language convention of starting them at 0.
A matrix in which all of the elements are real numbers is called a real matrix. A matrix that contains at least one complex number is called a complex matrix. A real or complex matrix $A$ with the property that $a(i,j) = a(j,i)$, is called a symmetric matrix. A complex matrix $A$ with the property that $a(i,j) = \text{conj}(a(j,i))$, is called a Hermitian matrix. Note that programs that manipulate symmetric and Hermitian matrices need only store half of the matrix values, since the values of the non-stored elements can be quickly reconstructed from the stored values.

A matrix that has the same number of rows as it has columns is referred to as a square matrix. The elements in a square matrix that have the same row index and column index are called the diagonal elements of the matrix, or simply the diagonal of the matrix.

The transpose of a matrix $A$ is the matrix obtained by “flipping” the elements of the array about its diagonal. That is, we exchange the elements $a(i,j)$ and $a(j,i)$. For a complex matrix, if we both flip the elements about the diagonal and then take the complex conjugate of the element, the resulting matrix is called the Hermitian transpose or conjugate transpose of the original matrix. The transpose and Hermitian transpose of a matrix $A$ are denoted by $A^T$ and $A^H$ respectively.

A column vector, or simply a vector, is a $n \times 1$ matrix, and a row vector is a $1 \times n$ matrix. A real or complex matrix $A$ is said to be positive definite if the vector-matrix product $x^TAx$ is greater than zero for all non-zero vectors $x$. A matrix that is not positive definite is referred to as indefinite.

An upper (or lower) triangular matrix, is a square matrix in which all elements below (or above) the diagonal are zero. A unit triangular matrix is an upper or lower triangular matrix with all 1’s along the diagonal.

A matrix $P$ is called a permutation matrix if, for any matrix $A$, the result of the matrix product $PA$ is identical to $A$ except for interchanging the rows of $A$. For a square matrix, it can be shown that if $PA$ is a permutation of the rows of $A$, then $P^T$ is the same permutation of the columns of $A$. Additionally, it can be shown that the inverse of $P$ is $P^T$.

In order to save space, a permutation matrix is usually stored as a linear array, called a permutation vector, rather than as an array. Specifically, if the permutation matrix maps the $i$-th row of a matrix to the $j$-th row, then the $i$-th element of the permutation vector is $j$.

A matrix with non-zero elements only on the diagonal is called a diagonal matrix. As is the case with a permutation matrix, it is usually stored as a vector of values, rather than as a matrix.

### Direct Method

For solvers that use the direct method, the basic technique employed in finding the solution of the system $Ax = b$ is to first factor $A$ into triangular matrices. That is, find a lower triangular matrix $L$ and an upper triangular matrix $U$, such that $A = LU$. Having obtained such a factorization (usually referred to as an $LU$ decomposition or $LU$ factorization), the solution to the original problem can be rewritten as follows.

\[
\begin{align*}
Ax &= b \\
LUx &= b \\
L(Ux) &= b
\end{align*}
\]

This leads to the following two-step process for finding the solution to the original system of equations:

1. Solve the systems of equations $Ly = b$.
2. Solve the system $UX = Y$.

Solving the systems $Ly = b$ and $UX = Y$ is referred to as a forward solve and a backward solve, respectively.

If a symmetric matrix $A$ is also positive definite, it can be shown that $A$ can be factored as $LL^T$ where $L$ is a lower triangular matrix. Similarly, a Hermitian matrix, $A$, that is positive definite can be factored as $A = LL^H$. For both symmetric and Hermitian matrices, a factorization of this form is called a Cholesky factorization.

In a Cholesky factorization, the matrix $U$ in an $LU$ decomposition is either $L^T$ or $L^H$. Consequently, a solver can increase its efficiency by only storing $L$, and one-half of $A$, and not computing $U$. Therefore, users who can express their application as the solution of a system of positive definite equations will gain a significant performance improvement over using a general representation.
For matrices that are symmetric (or Hermitian) but not positive definite, there are still some significant efficiencies to be had. It can be shown that if $A$ is symmetric but not positive definite, then $A$ can be factored as $A = LDL^T$, where $D$ is a diagonal matrix and $L$ is a lower unit triangular matrix. Similarly, if $A$ is Hermitian, it can be factored as $A = LDL^H$. In either case, we again only need to store $L$, $D$, and half of $A$ and we need not compute $U$. However, the backward solve phases must be amended to solving $L^T x = D^{-1} y$ rather than $L^T x = y$.

**Fill-In and Reordering of Sparse Matrices**

Two important concepts associated with the solution of sparse systems of equations are fill-in and reordering. The following example illustrates these concepts.

Consider the system of linear equation $Ax = b$, where $A$ is a symmetric positive definite sparse matrix, and $A$ and $b$ are defined by the following:

$$A = \begin{bmatrix}
9 & 3 & 0 & 0 \\
3 & 2 & 1 & 0 \\
0 & 1 & 2 & 0 \\
0 & 0 & 0 & 3 \\
\end{bmatrix}, \quad b = \begin{bmatrix}
1 \\
2 \\
3 \\
4 \\
\end{bmatrix}$$

A star (*) is used to represent zeros and to emphasize the sparsity of $A$. The Cholesky factorization of $A$ is: $A = LL^T$, where $L$ is the following:

$$L = \begin{bmatrix}
3 & * & * & * \\
1 & 1 & * & * \\
2 & 2 & * & * \\
1 & 1 & 1 & 1 \\
4 & -4 & -2 & 2 \\
1 & -1 & -2 & -3 \\
\end{bmatrix}$$

Notice that even though the matrix $A$ is relatively sparse, the lower triangular matrix $L$ has no zeros below the diagonal. If we computed $L$ and then used it for the forward and backward solve phase, we would do as much computation as if $A$ had been dense.

The situation of $L$ having non-zeros in places where $A$ has zeros is referred to as fill-in. Computationally, it would be more efficient if a solver could exploit the non-zero structure of $A$ in such a way as to reduce the fill-in when computing $L$. By doing this, the solver would only need to compute the non-zero entries in $L$.

Toward this end, consider permuting the rows and columns of $A$. As described in Matrix Fundamentals section, the permutations of the rows of $A$ can be represented as a permutation matrix, $P$. The result of permuting the rows is the product of $P$ and $A$. Suppose, in the above example, we swap the first and fifth row of $A$, then swap the first and fifth columns of $A$, and call the resulting matrix $B$. Mathematically, we can express the process of permuting the rows and columns of $A$ to get $B$ as $B = PAP^T$. After permuting the rows and columns of $A$, we see that $B$ is given by the following:
Since $B$ is obtained from $A$ by simply switching rows and columns, the numbers of non-zero entries in $A$ and $B$ are the same. However, when we find the Cholesky factorization, $B = LL^T$, we see the following:

The fill-in associated with $B$ is much smaller than the fill-in associated with $A$. Consequently, the storage and computation time needed to factor $B$ is much smaller than to factor $A$. Based on this, we see that an efficient sparse solver needs to find permutation $P$ of the matrix $A$, which minimizes the fill-in for factoring $B = PAP^T$, and then use the factorization of $B$ to solve the original system of equations.

Although the above example is based on a symmetric positive definite matrix and a Cholesky decomposition, the same approach works for a general $LU$ decomposition. Specifically, let $P$ be a permutation matrix, $B = PAP^T$ and suppose that $B$ can be factored as $B = LU$. Then

$$Ax = b$$

$$\Rightarrow PA(P^{-1}P)x = Pb$$
$$\Rightarrow PA(P^TP)x = Pb$$
$$\Rightarrow (PAP^T)(P^{-1}P)x = Pb$$
$$\Rightarrow B(Px) = Pb$$
$$\Rightarrow LU(Px) = Pb$$

It follows that if we obtain an $LU$ factorization for $B$, we can solve the original system of equations by a three step process:
1. Solve $Ly = Pb$.
2. Solve $Uz = y$.
3. Set $x = P^Tz$.

If we apply this three-step process to the current example, we first need to perform the forward solve of the systems of equation $Ly = Pb$:

$$Ly = \begin{bmatrix}
4 & * & * & * & * \\
* & \frac{1}{\sqrt{2}} & * & * & * \\
* & * & 2(\sqrt{3}) & * & * \\
* & * & * & \frac{\sqrt{10}}{4} & * \\
\frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{5}}{4}
\end{bmatrix} \begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5
\end{bmatrix} = \begin{bmatrix}
5 \\
2 \\
3 \\
4 \\
1
\end{bmatrix}$$

This gives:

$$y^T = \frac{5}{4}, 2\sqrt{2}, \frac{\sqrt{5}}{2}, \frac{16}{\sqrt{10}}, \frac{-979}{12}.$$ 

The second step is to perform the backward solve, $Uz = y$. Or, in this case, since a Cholesky factorization is used, $L^Tz = y$.

$$L^Tz = \begin{bmatrix}
4 & * & * & * & * \\
* & \frac{1}{\sqrt{2}} & * & * & * \\
* & * & 2(\sqrt{3}) & * & * \\
* & * & * & \frac{\sqrt{10}}{4} & * \\
\frac{3}{4} & \frac{3}{\sqrt{2}} & \sqrt{3} & \frac{3}{\sqrt{10}} & \frac{\sqrt{5}}{4}
\end{bmatrix} \begin{bmatrix}
z_1 \\
z_2 \\
z_3 \\
z_4 \\
z_5
\end{bmatrix} = \begin{bmatrix}
\frac{5}{4} \\
2(\sqrt{2}) \\
\frac{\sqrt{5}}{2} \\
\frac{16}{\sqrt{10}} \\
\frac{-979}{12}
\end{bmatrix}$$

This gives $z^T = \frac{123}{2}, \frac{983}{12}, \frac{1961}{12}, 398, \frac{-979}{3}$.

The third and final step is to set $x = P^Tz$. This gives $x^T = \frac{-979}{3}, \frac{983}{12}, \frac{1961}{12}, 398, \frac{123}{2}$. 

2525
Sparse Matrix Storage Formats

It is more efficient to store only the non-zero elements of a sparse matrix. There are a number of common
storage formats used for sparse matrices, but most of them employ the same basic technique. That is, store
all non-zero elements of the matrix into a linear array and provide auxiliary arrays to describe the locations
of the non-zero elements in the original matrix.

Storage Formats for the Direct Sparse Solvers

Storing the non-zero elements of a sparse matrix into a linear array is done by walking down each column
(column-major format) or across each row (row-major format) in order, and writing the non-zero elements to
a linear array in the order they appear in the walk.

- DSS Symmetric Matrix Storage
- DSS Nonsymmetric Matrix Storage
- DSS Structurally Symmetric Matrix Storage
- DSS Distributed Symmetric Matrix Storage

Sparse Matrix Storage Formats for Sparse BLAS Levels 2 and Level 3

These sections describe in detail the sparse matrix storage formats supported in the current version of the
Intel® MKL Sparse BLAS Level 2 and Level 3.

- Sparse BLAS CSR Matrix Storage
- Sparse BLAS CSC Matrix Storage
- Sparse BLAS Coordinate Matrix Storage
- Sparse BLAS Diagonal Matrix Storage
- Sparse BLAS Skyline Matrix Storage
- Sparse BLAS BSR Matrix Storage

DSS Symmetric Matrix Storage

For symmetric matrices, it is necessary to store only the upper triangular half of the matrix (upper triangular
format) or the lower triangular half of the matrix (lower triangular format).

The Intel® MKL direct sparse solvers use a row-major upper triangular storage format: the matrix is
compressed row-by-row and for symmetric matrices only non-zero elements in the upper triangular half of
the matrix are stored.

The Intel® MKL sparse matrix storage format for direct sparse solvers is specified by three arrays: values,
columns, and rowIndex. The following table describes the arrays in terms of the values, row, and column
positions of the non-zero elements in a sparse matrix.

<table>
<thead>
<tr>
<th>Array</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>A real or complex array that contains the non-zero elements of a sparse matrix. The non-zero elements are mapped into the values array using the row-major upper triangular storage mapping described above.</td>
</tr>
<tr>
<td>columns</td>
<td>Element i of the integer array columns is the number of the column that contains the i-th element in the values array.</td>
</tr>
<tr>
<td>rowIndex</td>
<td>Element j of the integer array rowIndex gives the index of the element in the values array that is first non-zero element in a row j.</td>
</tr>
</tbody>
</table>

The length of the values and columns arrays is equal to the number of non-zero elements in the matrix.
As the rowIndex array gives the location of the first non-zero element within a row, and the non-zero
elements are stored consecutively, the number of non-zero elements in the i-th row is equal to the difference of rowIndex[i] and rowIndex[i+1].
To have this relationship hold for the last row of the matrix, an additional entry (dummy entry) is added to the end of `rowIndex`. Its value is equal to the number of non-zero elements plus one. This makes the total length of the `rowIndex` array one larger than the number of rows in the matrix.

**NOTE**
The Intel® MKL sparse storage scheme for the direct sparse solvers supports both one-based indexing and zero-based indexing.

Consider the symmetric matrix $A$:

$$
A = \begin{pmatrix}
1 & -1 & * & -3 & * \\
-1 & 5 & * & * & * \\
* & 4 & 6 & 4 & * \\
-3 & * & 6 & 7 & * \\
* & 4 & * & -5 & *
\end{pmatrix}
$$

Only elements from the upper triangle are stored. The actual arrays for the matrix $A$ are as follows:

### Storage Arrays for a Symmetric Matrix

#### one-based indexing
- **values**
  
  \[
  (1, -1, -3, 5, 4, 6, 4, 7, -5)
  \]
- **columns**
  
  \[
  (1, 2, 4, 2, 3, 4, 5, 4, 5)
  \]
- **rowIndex**
  
  \[
  (1, 4, 5, 8, 9, 10)
  \]

#### zero-based indexing
- **values**
  
  \[
  (1, -1, -3, 5, 4, 6, 4, 7, -5)
  \]
- **columns**
  
  \[
  (0, 1, 3, 1, 2, 3, 4, 3, 4)
  \]
- **rowIndex**
  
  \[
  (0, 3, 4, 7, 8, 9)
  \]

### Storage Format Restrictions

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the `values` array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the `values` array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the `values` array.

### DSS Nonsymmetric Matrix Storage

For a non-symmetric or non-Hermitian matrix, all non-zero elements need to be stored. Consider the non-symmetric matrix $B$:

$$
B = \begin{pmatrix}
1 & -1 & * & -3 & * \\
-2 & 5 & * & * & * \\
* & 4 & 6 & 4 & * \\
-4 & * & 2 & 7 & * \\
* & 8 & * & -5 & *
\end{pmatrix}
$$

The matrix $B$ has 13 non-zero elements, and all of them are stored as follows:

### Storage Arrays for a Non-Symmetric Matrix

#### one-based indexing
- **values**
  
  \[
  (1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5)
  \]
- **columns**
  
  \[
  (1, 2, 4, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5)
  \]
Storage Format Restrictions
The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the values array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the values array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the values array.

DSS Structurally Symmetric Matrix Storage
Direct sparse solvers can also solve symmetrically structured systems of equations. A symmetrically structured system of equations is one where the pattern of non-zero elements is symmetric. That is, a matrix has a symmetric structure if \( a_{ij} \) is not zero if and only if \( a_{ji} \) is not zero. From the point of view of the solver software, a "non-zero" element of a matrix is any element stored in the values array, even if its value is equal to 0. In that sense, any non-symmetric matrix can be turned into a symmetrically structured matrix by carefully adding zeros to the values array. For example, the above matrix \( B \) can be turned into a symmetrically structured matrix by adding two non-zero entries:

\[
B = \begin{bmatrix}
  1 & -1 & 3 & * \\
  -2 & 5 & * & 0 \\
  * & * & 4 & 6 & 4 \\
  -4 & * & 2 & 7 & * \\
  * & 8 & 0 & * & -5
\end{bmatrix}
\]

The matrix \( B \) can be considered to be symmetrically structured with 15 non-zero elements and represented as:

Storage Arrays for a Symmetrically Structured Matrix

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the values array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the values array for any symmetric or structurally symmetric matrix.
The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the values array.

**DSS Distributed Symmetric Matrix Storage**

The distributed assembled matrix input format can be used by the Parallel Direct Sparse Solver for Clusters Interface.

In this format, the symmetric input matrix $A$ is divided into sequential row subsets, or domains. Each domain belongs to an MPI process. Neighboring domains can overlap. For such intersection between two domains, the element values of the full matrix can be obtained by summing the respective elements of both domains.

As in the centralized format, the distributed format uses three arrays to describe the input data, but the values, columns, and rowIndex arrays on each processor only describe the domain belonging to that particular processor and not the entire matrix.

For example, consider a symmetric matrix $A$:

$$
A = \begin{pmatrix}
6 & -1 & * & -3 & *\\
-1 & 5 & * & * & * \\
* & * & 11 & 5 & 4 \\
-3 & * & 5 & 10 & * \\
* & * & 4 & 5 & *
\end{pmatrix}
$$

This array could be distributed between two domains corresponding to two MPI processes, with the first containing rows 1 through 3, and the second containing rows 3 through 5.

**NOTE**

For the symmetric input matrix, it is not necessary to store the values from the lower triangle.

$$
A_{\text{Domain1}} = \begin{pmatrix}
6 & -1 & * & -3 & *\\
-1 & 5 & * & * & * \\
* & * & 3 & * & 2
\end{pmatrix}
$$

**Distributed Storage Arrays for a Symmetric Matrix, Domain 1**

<table>
<thead>
<tr>
<th>indexing</th>
<th>values</th>
<th>columns</th>
<th>rowIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-based indexing</td>
<td>(6, -1, -3, 5, 3, 2)</td>
<td>(1, 2, 4, 2, 3, 5)</td>
<td>(1, 4, 5, 7)</td>
</tr>
<tr>
<td>zero-based indexing</td>
<td>(6, -1, -3, 5, 3, 2)</td>
<td>(0, 1, 3, 1, 2, 4)</td>
<td>(0, 3, 4, 6)</td>
</tr>
</tbody>
</table>

$$
A_{\text{Domain2}} = \begin{pmatrix}
* & 8 & 5 & 2 \\
-3 & 5 & 10 & * \\
* & * & 4 & 5
\end{pmatrix}
$$

**Distributed Storage Arrays for a Symmetric Matrix, Domain 2**

<table>
<thead>
<tr>
<th>indexing</th>
<th>values</th>
<th>columns</th>
<th>rowIndex</th>
</tr>
</thead>
<tbody>
<tr>
<td>one-based indexing</td>
<td>(8, 5, 2, 10, 5)</td>
<td>(3, 4, 5, 4, 5)</td>
<td>(1, 4, 5, 6)</td>
</tr>
<tr>
<td>zero-based indexing</td>
<td>(8, 5, 2, 10, 5)</td>
<td>(2, 3, 4, 3, 4)</td>
<td>(0, 3, 4, 5)</td>
</tr>
</tbody>
</table>
The third row of matrix \( A \) is common between domain 1 and domain 2. The values of row 3 of matrix \( A \) are the sums of the respective elements of row 3 of matrix \( A_{\text{Domain1}} \) and row 1 of matrix \( A_{\text{Domain2}} \).

**Storage Format Restrictions**

The storage format for the sparse solver must conform to two important restrictions:

- the non-zero values in a given row must be placed into the \( \text{values} \) array in the order in which they occur in the row (from left to right);
- no diagonal element can be omitted from the \( \text{values} \) array for any symmetric or structurally symmetric matrix.

The second restriction implies that if symmetric or structurally symmetric matrices have zero diagonal elements, then they must be explicitly represented in the \( \text{values} \) array.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**Sparse BLAS CSR Matrix Storage Format**

The Intel® MKL Sparse BLAS compressed sparse row (CSR) format is specified by four arrays:

- \( \text{values} \)
- \( \text{columns} \)
- \( \text{pointerB} \)
- \( \text{pointerE} \)

In addition, each sparse matrix has an associated variable, \( \text{indexing} \), which specifies if the matrix indices are 0-based (\( \text{indexing}=0 \)) or 1-based (\( \text{indexing}=1 \)). These are descriptions of the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix \( A \).

- \( \text{values} \): A real or complex array that contains the non-zero elements of \( A \). Values of the non-zero elements of \( A \) are mapped into the \( \text{values} \) array using the row-major storage mapping described above.

- \( \text{columns} \): Element \( i \) of the integer array \( \text{columns} \) is the number of the column in \( A \) that contains the \( i \)-th value in the \( \text{values} \) array.

- \( \text{pointerB} \): Element \( j \) of this integer array gives the index of the element in the \( \text{values} \) array that is first non-zero element in a row \( j \) of \( A \). Note that this index is equal to \( \text{pointerB}[j]-\text{indexing} \).

- \( \text{pointerE} \): An integer array that contains row indices, such that \( \text{pointerE}[j]-\text{indexing} \) is the index of the element in the \( \text{values} \) array that is last non-zero element in a row \( j \) of \( A \).

The length of the \( \text{values} \) and \( \text{columns} \) arrays is equal to the number of non-zero elements in \( A \). The length of the \( \text{pointerB} \) and \( \text{pointerE} \) arrays is equal to the number of rows in \( A \).
Note that the Intel® MKL Sparse BLAS routines support the CSR format both with one-based indexing and zero-based indexing.

You can represent the matrix $B$

\[
B = \begin{pmatrix}
1 & -1 & -3 & * \\
-2 & 5 & * & *
\end{pmatrix}
\]

in the CSR format as:

### Storage Arrays for a Matrix in CSR Format

#### one-based indexing

| values | 1 -1 -3 -2 5 4 6 4 -4 2 7 8 -5 |
| columns | 1 2 4 1 2 3 4 5 1 3 4 2 5 |
| pointerB | 1 2 4 1 2 3 4 5 1 3 4 2 5 |
| pointerE | 4 6 9 12 14 |

#### zero-based indexing

| values | 1 -1 -3 -2 5 4 6 4 -4 2 7 8 -5 |
| columns | 0 1 3 0 1 2 3 4 0 2 3 1 4 |
| pointerB | 0 3 5 8 11 |
| pointerE | 4 6 9 12 14 |

Additionally, you can define submatrices with different pointerB and pointerE arrays that share the same values and columns arrays of a CSR matrix. For example, you can represent the lower right 3x3 submatrix of $B$ as:

### Storage Arrays for a Matrix in CSR Format

#### one-based indexing

| subpointerB | 6 10 13 |
| subpointerE | 9 12 14 |

#### zero-based indexing

| subpointerB | 5 9 12 |
| subpointerE | 8 11 13 |

The CSR matrix must have a monotonically increasing row index. That is, $pointerB[i] \leq pointerB[j]$ and $pointerE[i] \leq pointerE[j]$ for all indices $i < j$.

This storage format is used in the NIST Sparse BLAS library [Rem05].

### Three Array Variation of CSR Format

The storage format accepted for the direct sparse solvers is a variation of the CSR format. It also is used in the Intel® MKL Sparse BLAS Level 2 both with one-based indexing and zero-based indexing. The above matrix $B$ can be represented in this format (referred to as the 3-array variation of the CSR format or CSR3) as:

### Storage Arrays for a Matrix in CSR Format (3-Array Variation)

#### one-based indexing

| values | 1 -1 -3 -2 5 4 6 4 -4 2 7 8 -5 |
| columns | 1 2 4 1 2 3 4 5 1 3 4 2 5 |
| rowIndex | 1 4 6 9 12 14 |
The 3-array variation of the CSR format has a restriction: all non-zero elements are stored continuously, that is the set of non-zero elements in the row \( J \) goes just after the set of non-zero elements in the row \( J-1 \).

There are no such restrictions in the general (NIST) CSR format. This may be useful, for example, if there is a need to operate with different submatrices of the matrix at the same time. In this case, it is enough to define the arrays \( \text{pointerB} \) and \( \text{pointerE} \) for each needed submatrix so that all these arrays are pointers to the same array \( \text{values} \).

By definition, the array \( \text{rowIndex} \) from the Table "Storage Arrays for a Non-Symmetric Example Matrix" is related to the arrays \( \text{pointerB} \) and \( \text{pointerE} \) from the Table "Storage Arrays for an Example Matrix in CSR Format", and you can see that:

\[
\text{pointerB}[i] = \text{rowIndex}[i] \quad \text{for } i = 0, ..4;
\]
\[
\text{pointerE}[i] = \text{rowIndex}[i+1] \quad \text{for } i = 0, ..4.
\]

This enables calling a routine that has \( \text{values} \), \( \text{columns} \), \( \text{pointerB} \) and \( \text{pointerE} \) as input parameters for a sparse matrix stored in the format accepted for the direct sparse solvers. For example, a routine with the interface:

\[
\text{void name_routine}(.... , \text{double *} \text{values}, \text{MKL_INT *} \text{columns}, \text{MKL_INT *} \text{pointerB}, \text{MKL_INT *} \text{pointerE}, ...)\]

can be called with parameters \( \text{values} \), \( \text{columns} \), \( \text{rowIndex} \) as follows:

\[
\text{name_routine}(.... , \text{values}, \text{columns}, \text{rowIndex}, \text{rowIndex+1}, ...).\]

**Sparse BLAS CSC Matrix Storage Format**

The compressed sparse column format (CSC) is similar to the CSR format, but the columns are used instead of the rows. In other words, the CSC format is identical to the CSR format for the transposed matrix. The CSR format is specified by four arrays: \( \text{values} \), \( \text{columns} \), \( \text{pointerB} \), and \( \text{pointerE} \). The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix \( A \).

- **values**: A real or complex array that contains the non-zero elements of \( A \). Values of the non-zero elements of \( A \) are mapped into the \( \text{values} \) array using the column-major storage mapping.
- **rows**: Element \( i \) of the integer array \( \text{rows} \) is the number of the row in \( A \) that contains the \( i \)-th value in the \( \text{values} \) array.
- **pointerB**: Element \( j \) of this integer array gives the index of the element in the \( \text{values} \) array that is first non-zero element in a column \( j \) of \( A \). Note that this index is equal to \( \text{pointerB}[j] \)-indexing for Inspector-executor Sparse BLAS CSC arrays.
- **pointerE**: An integer array that contains column indices, such that \( \text{pointerE}[j] \)-indexing is the index of the element in the \( \text{values} \) array that is last non-zero element in a column \( j \) of \( A \).

The length of the \( \text{values} \) and \( \text{columns} \) arrays is equal to the number of non-zero elements in \( A \). The length of the \( \text{pointerB} \) and \( \text{pointerE} \) arrays is equal to the number of columns in \( A \).
NOTE
Note that the Intel® MKL Sparse BLAS routines support the CSC format both with one-based indexing and zero-based indexing.

For example, consider matrix $B$:

$$
B = \begin{pmatrix}
1 & -1 & 3 & * \\
-2 & 5 & * & * \\
* & 4 & 6 & 4 \\
* & 2 & 7 & * \\
* & 8 & * & -5
\end{pmatrix}
$$

It can be represented in the CSC format as:

**Storage Arrays for a Matrix in CSC Format**

<table>
<thead>
<tr>
<th>one-based indexing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{values}$</td>
<td>$(1, -2, -4, -1, 5, 8, 4, 2, -3, 6, 7, 4, -5)$</td>
</tr>
<tr>
<td>$\text{rows}$</td>
<td>$(1, 2, 4, 1, 2, 5, 3, 4, 1, 3, 4, 3, 5)$</td>
</tr>
<tr>
<td>$\text{pointerB}$</td>
<td>$(1, 4, 7, 9, 12)$</td>
</tr>
<tr>
<td>$\text{pointerE}$</td>
<td>$(4, 7, 9, 12, 14)$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>zero-based indexing</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{values}$</td>
<td>$(1, -2, -4, -1, 5, 8, 4, 2, -3, 6, 7, 4, -5)$</td>
</tr>
<tr>
<td>$\text{rows}$</td>
<td>$(0, 1, 3, 0, 1, 4, 2, 3, 0, 2, 3, 2, 4)$</td>
</tr>
<tr>
<td>$\text{pointerB}$</td>
<td>$(0, 3, 6, 8, 11)$</td>
</tr>
<tr>
<td>$\text{pointerE}$</td>
<td>$(3, 6, 8, 11, 13)$</td>
</tr>
</tbody>
</table>

**Sparse BLAS Coordinate Matrix Storage Format**

The coordinate format is the most flexible and simplest format for the sparse matrix representation. Only non-zero elements are stored, and the coordinates of each non-zero element are given explicitly. Many commercial libraries support the matrix-vector multiplication for the sparse matrices in the coordinate format.

The Intel® MKL coordinate format is specified by three arrays: $\text{values}$, $\text{rows}$, and $\text{columns}$, and a parameter $\text{nnz}$ which is number of non-zero elements in $A$. All three arrays have dimension $\text{nnz}$. The following table describes the arrays in terms of the values, row, and column positions of the non-zero elements in a sparse matrix $A$.

- **values**: A real or complex array that contains the non-zero elements of $A$ in any order.
- **rows**: Element $i$ of the integer array $\text{rows}$ is the number of the row in $A$ that contains the $i$-th value in the $\text{values}$ array.
- **columns**: Element $i$ of the integer array $\text{columns}$ is the number of the column in $A$ that contains the $i$-th value in the $\text{values}$ array.

**NOTE**
Note that the Intel® MKL Sparse BLAS routines support the coordinate format both with one-based indexing and zero-based indexing.

For example, the sparse matrix $C$

$$
C = \begin{pmatrix}
1 & -1 & -3 & 0 & 0 \\
-2 & 5 & 0 & 0 & 0 \\
0 & 0 & 4 & 6 & 4 \\
-4 & 0 & 2 & 7 & 0 \\
0 & 8 & 0 & 0 & -5
\end{pmatrix}
$$
can be represented in the coordinate format as follows:

**Storage Arrays for an Example Matrix in case of the coordinate format**

<table>
<thead>
<tr>
<th>one-based indexing</th>
<th>zero-based indexing</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</td>
<td>values</td>
</tr>
<tr>
<td>rows</td>
<td>rows</td>
</tr>
<tr>
<td>columns</td>
<td>columns</td>
</tr>
</tbody>
</table>

values = \((1, -1, -3, -2, 5, 4, 6, 4, -4, 2, 7, 8, -5)\)

rows = \((1, 1, 1, 2, 2, 3, 3, 3, 4, 4, 4, 5, 5)\)

columns = \((1, 2, 3, 1, 2, 3, 4, 5, 1, 3, 4, 2, 5)\)

Sparse BLAS Diagonal Matrix Storage Format

If the sparse matrix has diagonals containing only zero elements, then the diagonal storage format can be used to reduce the amount of information needed to locate the non-zero elements. This storage format is particularly useful in many applications where the matrix arises from a finite element or finite difference discretization. The Intel® MKL diagonal storage format is specified by two arrays: \(values\) and \(distance\), and two parameters: \(ndiag\), which is the number of non-empty diagonals, and \(lval\), which is the declared leading dimension in the calling (sub)programs. The following table describes the arrays \(values\) and \(distance\):

**values**

A real or complex two-dimensional array is dimensioned as \(lval\) by \(ndiag\). Each column of it contains the non-zero elements of certain diagonal of \(A\). The key point of the storage is that each element in \(values\) retains the row number of the original matrix. To achieve this diagonals in the lower triangular part of the matrix are padded from the top, and those in the upper triangular part are padded from the bottom. Note that the value of \(distance[i]\) is the number of elements to be padded for diagonal \(i\).

**distance**

An integer array with dimension \(ndiag\). Element \(i\) of the array \(distance\) is the distance between \(i\)-diagonal and the main diagonal. The distance is positive if the diagonal is above the main diagonal, and negative if the diagonal is below the main diagonal. The main diagonal has a distance equal to zero.

The above matrix \(C\) can be represented in the diagonal storage format as follows:

\[
\begin{align*}
\text{distance} & = (-3, -1, 0, 1, 2) \\
\text{values} & = \begin{bmatrix}
* & * & 1 & -1 & -3 \\
* & -2 & 5 & 0 & 0 \\
* & 0 & 4 & 6 & 4 \\
-4 & 2 & 7 & 0 & * \\
8 & 0 & -5 & * & *
\end{bmatrix}
\end{align*}
\]

where the asterisks denote padded elements.

When storing symmetric, Hermitian, or skew-symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For the Intel® MKL triangular solver routines elements of the array \(distance\) must be sorted in increasing order. In all other cases the diagonals and distances can be stored in arbitrary order.

Sparse BLAS Skyline Matrix Storage Format

The skyline storage format is important for the direct sparse solvers, and it is well suited for Cholesky or LU decomposition when no pivoting is required.

The skyline storage format accepted in Intel® MKL can store only triangular matrix or triangular part of a matrix. This format is specified by two arrays: \(values\) and \(pointers\). The following table describes these arrays:
values
A scalar array. For a lower triangular matrix it contains the set of elements from each row of the matrix starting from the first non-zero element to and including the diagonal element. For an upper triangular matrix it contains the set of elements from each column of the matrix starting with the first non-zero element down to and including the diagonal element. Encountered zero elements are included in the sets.

pointers
An integer array with dimension \((m+1)\), where \(m\) is the number of rows for lower triangle (columns for the upper triangle). \(\text{pointers}[i] - \text{pointers}[0]+1\) gives the index of element in values that is first non-zero element in row (column) \(i\). The value of \(\text{pointers}[m]\) is set to \(\text{nnz}+\text{pointers}[0]\), where \(\text{nnz}\) is the number of elements in the array values.

For example, consider the matrix \(C\):
\[
C = \begin{bmatrix}
1 & -1 & 0 & -3 & 0 & 0 \\
-2 & 5 & 0 & 0 & 0 & 0 \\
0 & 0 & 4 & 6 & 0 & 0 \\
-4 & 0 & 2 & 7 & 0 & 0 \\
0 & 8 & 0 & 0 & 0 & -5 \\
\end{bmatrix}
\]
The low triangle of the matrix \(C\) given above can be stored as follows:
\[
\begin{array}{c}
\text{values} = [ 1 \ -2 \ 5 \ 4 \ -4 \ 0 \ 2 \ 7 \ 8 \ 0 \ 0 \ -5 ] \\
\text{pointers} = [ 0 \ 1 \ 3 \ 4 \ 8 \ 12 ]
\end{array}
\]
and the upper triangle of this matrix \(C\) can be stored as follows:
\[
\begin{array}{c}
\text{values} = [ 1 \ -1 \ 5 \ -3 \ 0 \ 4 \ 6 \ 7 \ 4 \ 0 \ -5 ] \\
\text{pointers} = [ 0 \ 1 \ 3 \ 6 \ 8 \ 11 ]
\end{array}
\]

This storage format is supported by the NIST Sparse BLAS library [Rem05]. Note that the Intel® MKL Sparse BLAS routines operating with the skyline storage format do not support general matrices.

Sparse BLAS BSR Matrix Storage Format
The Intel® MKL block compressed sparse row (BSR) format for sparse matrices is specified by four arrays: values, columns, pointerB, and pointerE. The following table describes these arrays.

values
A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block-by-block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them are equal to zero. Within each non-zero block elements are stored in column-major order in the case of one-based indexing, and in row-major order in the case of the zero-based indexing.

columns
Element \(i\) of the integer array columns is the number of the column in the block matrix that contains the \(i\)-th non-zero block.

pointerB
Element \(j\) of this integer array gives the index of the element in the columns array that is first non-zero block in a row \(j\) of the block matrix.

pointerE
Element \(j\) of this integer array gives the index of the element in the columns array that contains the last non-zero block in a row \(j\) of the block matrix plus 1.

The length of the values array is equal to the number of all elements in the non-zero blocks, the length of the columns array is equal to the number of non-zero blocks. The length of the pointerB and pointerE arrays is equal to the number of block rows in the block matrix.
NOTE
Note that the Intel® MKL Sparse BLAS routines support BSR format both with one-based indexing and zero-based indexing.

For example, consider the sparse matrix $D$

\[
\begin{bmatrix}
1 & 0 & 6 & 7 & * & * \\
2 & 1 & 8 & 2 & * & * \\
* & * & 1 & 4 & * & * \\
* & * & 5 & 1 & * & * \\
* & * & 4 & 3 & 7 & 2 \\
* & * & 0 & 0 & 0 & 0
\end{bmatrix}
\]

If the size of the block equals 2, then the sparse matrix $D$ can be represented as a 3x3 block matrix $E$ with the following structure:

\[
E = \begin{bmatrix}
L & M & * \\
* & N & * \\
* & P & Q
\end{bmatrix}
\]

where

\[
L = \begin{bmatrix} 1 & 0 \\
2 & 1 
\end{bmatrix},
M = \begin{bmatrix} 6 & 7 \\
8 & 2 
\end{bmatrix},
N = \begin{bmatrix} 4 & 1 \\
5 & 1 
\end{bmatrix},
P = \begin{bmatrix} 4 & 3 \\
0 & 0 
\end{bmatrix},
Q = \begin{bmatrix} 7 & 2 \\
0 & 0 
\end{bmatrix}
\]

The matrix $D$ can be represented in the BSR format as follows:

one-based indexing

| values | (1 2 0 1 6 8 7 2 1 5 4 1 4 0 3 0 7 0 2 0) |
| columns | (1 2 2 2 3) |
| pointerB | (1 3 4) |
| pointerE | (3 4 6) |

described by the NIST Sparse BLAS library [Rem05].

Three Array Variation of BSR Format

Intel® MKL supports the variation of the BSR format that is specified by three arrays: values, columns, and rowIndex. The following table describes these arrays.

values

A real array that contains the elements of the non-zero blocks of a sparse matrix. The elements are stored block by block in row-major order. A non-zero block is the block that contains at least one non-zero element. All elements of non-zero blocks are stored, even if some of them is equal to zero. Within each non-zero block the elements are stored in column major order in the case of the one-based indexing, and in row major order in the case of the zero-based indexing.

columns

Element $i$ of the integer array columns is the number of the column in the block matrix that contains the $i$-th non-zero block.

rowIndex

Element $j$ of this integer array gives the index of the element in the columns array that is first non-zero block in a row $j$ of the block matrix.
The length of the \textit{values} array is equal to the number of all elements in the non-zero blocks, the length of the \textit{columns} array is equal to the number of non-zero blocks.

As the \textit{rowIndex} array gives the location of the first non-zero block within a row, and the non-zero blocks are stored consecutively, the number of non-zero blocks in the \textit{i}-th row is equal to the difference of \textit{rowIndex}[\textit{i}] and \textit{rowIndex}[\textit{i}+1].

To retain this relationship for the last row of the block matrix, an additional entry (dummy entry) is added to the end of \textit{rowIndex} with value equal to the number of non-zero blocks plus one. This makes the total length of the \textit{rowIndex} array one larger than the number of rows of the block matrix.

The above matrix \(D\) can be represented in this 3-array variation of the BSR format as follows:

\begin{verbatim}
one-based indexing
values = [1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0]
columns = [1 2 2 2 3]
rowIndex = [1 3 4 6]
\end{verbatim}

\begin{verbatim}
zero-based indexing
values = [1 0 2 1 6 7 8 2 1 4 5 1 4 3 0 0 7 2 0 0]
columns = [0 1 1 1 2]
rowIndex = [0 2 3 5]
\end{verbatim}

When storing symmetric matrices, it is necessary to store only the upper or the lower triangular part of the matrix.

For example, consider the symmetric sparse matrix \(F\):

\[
F = \begin{pmatrix}
1 & 0 & 6 & 7 & * & * \\
2 & 1 & 8 & 2 & * & * \\
6 & 8 & 1 & 4 & * & * \\
7 & 2 & 5 & 2 & * & * \\
* & * & * & 7 & 2 \\
* & * & * & 0 & 0
\end{pmatrix}
\]

If the size of the block equals 2, then the sparse matrix \(F\) can be represented as a 3x3 block matrix \(G\) with the following structure:

\[
G = \begin{pmatrix}
L & M & * \\
M' & N & * \\
* & * & Q
\end{pmatrix}
\]

where

\[
L = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \quad M = \begin{pmatrix} 6 & 7 \\ 8 & 2 \end{pmatrix}, \quad M' = \begin{pmatrix} 6 & 8 \\ 7 & 2 \end{pmatrix}, \quad N = \begin{pmatrix} 1 & 4 \\ 5 & 2 \end{pmatrix}, \quad \text{and} \quad Q = \begin{pmatrix} 7 & 2 \\ 0 & 0 \end{pmatrix}
\]

The symmetric matrix \(F\) can be represented in this 3-array variation of the BSR format (storing only the upper triangular part) as follows:

\begin{verbatim}
one-based indexing
values = [1 2 0 1 6 8 7 2 1 5 4 2 4 0 3 0 7 0 2 0]
columns = [1 2 2 3]
rowIndex = [1 3 4 6]
\end{verbatim}

\begin{verbatim}
zero-based indexing
values = [1 0 2 1 6 7 8 2 1 4 5 2 7 2 0 0]
columns = [0 1 1 2]
rowIndex = [0 2 3 4]
\end{verbatim}
Variable BSR Format
A variation of BSR3 is variable block compressed sparse row format. For a trust level \( t \), \( 0 \leq t \leq 100 \), rows similar up to \( t \) percent are placed in one supernode.
The major arguments in the BLAS routines are vector and matrix, whereas VM functions work on vector arguments only. The sections that follow discuss each of these arguments and provide examples.

**Vector Arguments in BLAS**

Vector arguments are passed in one-dimensional arrays. The array dimension (length) and vector increment are passed as integer variables. The length determines the number of elements in the vector. The increment (also called stride) determines the spacing between vector elements and the order of the elements in the array in which the vector is passed.

A vector of length \( n \) and increment \( \text{incx} \) is passed in a one-dimensional array \( x \) whose values are defined as

\[
x[0], x[|\text{incx}|], \ldots, x[(n-1)*|\text{incx}|]
\]

If \( \text{incx} \) is positive, then the elements in array \( x \) are stored in increasing order. If \( \text{incx} \) is negative, the elements in array \( x \) are stored in decreasing order with the first element defined as \( x[(n-1)*|\text{incx}|] \). If \( \text{incx} \) is zero, then all elements of the vector have the same value, \( x[0] \). The size of the one-dimensional array that stores the vector must always be at least

\[
\text{idimx} = 1 + (n-1)*|\text{incx}|
\]

**Example. One-dimensional Real Array**

Let \( x[0:6] \) be the one-dimensional real array

\[
x = [1.0, 3.0, 5.0, 7.0, 9.0, 11.0, 13.0].
\]

If \( \text{incx} = 2 \) and \( n = 3 \), then the vector argument with elements in order from first to last is \([1.0, 5.0, 9.0] \).

If \( \text{incx} = -2 \) and \( n = 4 \), then the vector elements in order from first to last is \([13.0, 9.0, 5.0, 1.0] \).

If \( \text{incx} = 0 \) and \( n = 4 \), then the vector elements in order from first to last is \([1.0, 1.0, 1.0, 1.0] \).

One-dimensional substructures of a matrix, such as the rows, columns, and diagonals, can be passed as vector arguments with the starting address and increment specified.

Storage of the \( m \)-by-\( n \) matrix can be based on either column-major ordering where the increment between elements in the same column is 1, the increment between elements in the same row is \( m \), and the increment between elements on the same diagonal is \( m + 1 \); or row-major ordering where the increment between elements in the same row is 1, the increment between elements in the same column is \( n \), and the increment between elements on the same diagonal is \( n + 1 \).

**Example. Two-dimensional Real Matrix**

Let \( a \) be a real 5 \( \times \) 4 matrix declared as .

To scale the third column of \( a \) by 2.0, use the BLAS routine \( \text{sscal} \) with the following calling sequence:

\[
\text{cblas_sscal}(5, 2.0, a[2], 4)
\]

To scale the second row, use the statement:

\[
\text{cblas_sscal}(4, 2.0, a[4], 1)
\]
To scale the main diagonal of $a$ by 2.0, use the statement:

```
cblas_sscal (4, 2.0, a[0], 5)
```

**NOTE**
The default vector argument is assumed to be 1.

## Vector Arguments in VM

Vector arguments of VM mathematical functions are passed in one-dimensional arrays with unit vector increment. It means that a vector of length $n$ is passed contiguously in an array $a$ whose values are defined as

$$a[0], a[1], \ldots, a[n-1].$$

To accommodate for arrays with other increments, or more complicated indexing, VM contains auxiliary pack/unpack functions that gather the array elements into a contiguous vector and then scatter them after the computation is complete.

Generally, if the vector elements are stored in a one-dimensional array $a$ as $a[m0], a[m1], \ldots, a[mn-1]$ and need to be regrouped into an array $y$ as $y[k0], y[k1], \ldots, y[kn-1],..$

VM pack/unpack functions can use one of the following indexing methods:

### Positive Increment Indexing

$$kj = incy * j, mj = inca * j, j = 0, \ldots, n-1.$$  
**Constraint:** $incy > 0$ and $inca > 0.$

For example, setting $incy = 1$ specifies gathering array elements into a contiguous vector. This method is similar to that used in BLAS, with the exception that negative and zero increments are not permitted.

### Index Vector Indexing

$$kj = iy[j], mj = ia[j], j = 0, \ldots, n-1.$$  
where $ia$ and $iy$ are arrays of length $n$ that contain index vectors for the input and output arrays $a$ and $y$, respectively.

### Mask Vector Indexing

Indices $kj, mj$ are such that:

$$my[kj] \neq 0, ma[mj] \neq 0, j = 0, \ldots, n-1.$$  
where $ma$ and $my$ are arrays that contain mask vectors for the input and output arrays $a$ and $y$, respectively.
Matrix Arguments

Matrix arguments of the Intel® Math Kernel Library routines can be stored in arrays, using the following storage schemes:

- conventional full storage
- packed storage for Hermitian, symmetric, or triangular matrices
- band storage for band matrices
- rectangular full packed storage for symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels.

Full storage is the simplest scheme. A matrix $A$ is stored in a one-dimensional array $a$, with the matrix element $a_{ij}$ stored in the array element $a[i-1 + (j-1)\times lda]$, where $lda$ is the leading dimension of array $a$.

If a matrix is triangular (upper or lower, as specified by the argument $uplo$), only the elements of the relevant triangle are stored; the remaining elements of the array need not be set.

Routines that handle symmetric or Hermitian matrices allow for either the upper or lower triangle of the matrix to be stored in the corresponding elements of the array:

if $uplo = 'U'$,

$a_{ij}$ is stored as described for $i \leq j$, other elements of $a$ need not be set.

if $uplo = 'L'$,

$a_{ij}$ is stored as described for $j \leq i$, other elements of $a$ need not be set.

Packed storage allows you to store symmetric, Hermitian, or triangular matrices more compactly: the relevant triangle (again, as specified by the argument $uplo$) is packed by columns in a one-dimensional array $ap$:

if $uplo = 'U'$, $a_{ij}$ is stored in $ap[i - 1 + j(j - 1)/2]$ for $i \leq j$

if $uplo = 'L'$, $a_{ij}$ is stored in $ap[i - 1 + (2\times n - j)\times (j - 1)/2]$ for $j \leq i$.

In descriptions of LAPACK routines, arrays with packed matrices have names ending in $p$.

Band storage is as follows: an $m$-by-$n$ band matrix with $kl$ non-zero sub-diagonals and $ku$ non-zero super-diagonals is stored compactly in an array $ab$ with $(kl+ku+1)\times n$ elements. Thus, $a_{ij}$ is stored in $ab(ku+1+i-j,j)$ for $max(1,j-ku) \leq i \leq min(n,j+kl)$.

Use the band storage scheme only when $kl$ and $ku$ are much less than the matrix size $n$. Although the routines work correctly for all values of $kl$ and $ku$, using the band storage is inefficient if your matrices are not really banded.

The band storage scheme is illustrated by the following example, when $m = n = 6, kl = 2, ku = 1$. Array elements marked * are not used by the routines:

<table>
<thead>
<tr>
<th>Banded matrix $A$</th>
<th>Band storage of $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11} \ a_{12} \ 0 \ 0 \ 0 \ 0$</td>
<td>$* \ a_{12} \ a_{23} \ a_{34} \ a_{45} \ a_{56}$</td>
</tr>
<tr>
<td>$a_{21} \ a_{22} \ a_{23} \ 0 \ 0 \ 0$</td>
<td>$a_{11} \ a_{22} \ a_{33} \ a_{44} \ a_{55} \ a_{66}$</td>
</tr>
<tr>
<td>$a_{31} \ a_{32} \ a_{33} \ a_{34} \ 0 \ 0$</td>
<td>$a_{21} \ a_{32} \ a_{43} \ a_{54} \ a_{65} \ *$</td>
</tr>
<tr>
<td>$0 \ a_{42} \ a_{43} \ a_{44} \ a_{45} \ 0$</td>
<td>$a_{31} \ a_{42} \ a_{53} \ a_{64} \ * \ *$</td>
</tr>
<tr>
<td>$0 \ 0 \ a_{53} \ a_{54} \ a_{55} \ a_{56}$</td>
<td></td>
</tr>
<tr>
<td>$0 \ 0 \ 0 \ a_{64} \ a_{65} \ a_{66}$</td>
<td></td>
</tr>
</tbody>
</table>
When a general band matrix is supplied for LU factorization, space must be allowed to store $kl$ additional super-diagonals generated by fill-in as a result of row interchanges. This means that the matrix is stored according to the above scheme, but with $kl + ku$ super-diagonals. Thus, $a_{ij}$ is stored in $ab(kl+ku+1+i-j,j)$ for $\max(1,j-ku) \leq i \leq \min(n,j+kl)$.

The band storage scheme for LU factorization is illustrated by the following example, when $m = n = 6$, $kl = 2$, $ku = 1$:

<table>
<thead>
<tr>
<th>Banded matrix $A$</th>
<th>Band storage of $A$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{11}$ $a_{12}$ 0 0 0 0</td>
<td>*  *  *  +  +  +</td>
</tr>
<tr>
<td>$a_{21}$ $a_{22}$ $a_{23}$ 0 0 0</td>
<td>*  *  +  +  +  +</td>
</tr>
<tr>
<td>$a_{31}$ $a_{32}$ $a_{33}$ $a_{34}$ 0 0</td>
<td>*  $a_{12}$ $a_{23}$ $a_{34}$ $a_{45}$ $a_{56}$</td>
</tr>
<tr>
<td>0 $a_{42}$ $a_{43}$ $a_{44}$ $a_{45}$ 0</td>
<td>$a_{11}$ $a_{22}$ $a_{33}$ $a_{44}$ $a_{55}$ $a_{66}$</td>
</tr>
<tr>
<td>0 0 $a_{53}$ $a_{54}$ $a_{55}$ $a_{56}$</td>
<td>$a_{21}$ $a_{32}$ $a_{43}$ $a_{54}$ $a_{65}$ *</td>
</tr>
<tr>
<td>0 0 0 $a_{64}$ $a_{65}$ $a_{66}$</td>
<td>$a_{31}$ $a_{42}$ $a_{53}$ $a_{64}$ * *</td>
</tr>
</tbody>
</table>

Array elements marked * are not used by the routines; elements marked + need not be set on entry, but are required by the LU factorization routines to store the results. The input array will be overwritten on exit by the details of the LU factorization as follows:

\[
\begin{align*}
* & * & * & u_{14} & u_{25} & u_{36} \\
* & * & u_{13} & u_{24} & u_{35} & u_{46} \\
* & u_{12} & u_{23} & u_{34} & u_{45} & u_{56} \\
u_{11} & u_{22} & u_{33} & u_{44} & u_{55} & u_{66} \\
m_{21} & m_{32} & m_{43} & m_{54} & m_{65} & * \\
m_{31} & m_{42} & m_{53} & m_{64} & * & *
\end{align*}
\]

where $u_{ij}$ are the elements of the upper triangular matrix $U$, and $m_{ij}$ are the multipliers used during factorization.

Triangular band matrices are stored in the same format, with either $kl = 0$ if upper triangular, or $ku = 0$ if lower triangular. For symmetric or Hermitian band matrices with $k$ sub-diagonals or super-diagonals, you need to store only the upper or lower triangle, as specified by the argument `uplo`:

- **if uplo = 'U',** $a_{ij}$ is stored in $ab(k+1+i-j,j)$ for $\max(1,j-k) \leq i \leq j$
- **if uplo = 'L',** $a_{ij}$ is stored in $ab(1+i-j,j)$ for $j \leq i \leq \min(n,j+k)$.

In descriptions of LAPACK routines, arrays that hold matrices in band storage have names ending in $b$.

In Fortran, column-major ordering of storage is assumed. This means that elements of the same column occupy successive storage locations.

Three quantities are usually associated with a two-dimensional array argument: its leading dimension, which specifies the number of storage locations between elements in the same row, its number of rows, and its number of columns. For a matrix in full storage, the leading dimension of the array must be at least as large as the number of rows in the matrix.

A character transposition parameter is often passed to indicate whether the matrix argument is to be used in normal or transposed form or, for a complex matrix, if the conjugate transpose of the matrix is to be used.

The values of the transposition parameter for these three cases are the following:
'N' or 'n'  
normal (no conjugation, no transposition)
'T' or 't'  
transpose
'C' or 'c'  
conjugate transpose.

Example. Two-Dimensional Complex Array

Suppose \( A(1:5, 1:4) \) is the complex two-dimensional array presented by matrix

\[
\begin{bmatrix}
(1.1, 0.11) & (1.2, 0.12) & (1.3, 0.13) & (1.4, 0.14) \\
(2.1, 0.21) & (2.2, 0.22) & (2.3, 0.23) & (1.4, 0.24) \\
(3.1, 0.31) & (3.2, 0.32) & (3.3, 0.33) & (1.4, 0.34) \\
(4.1, 0.41) & (4.2, 0.42) & (4.3, 0.43) & (1.4, 0.44) \\
(5.1, 0.51) & (5.2, 0.52) & (5.3, 0.53) & (1.4, 0.54)
\end{bmatrix}
\]

Let \( \text{transa} \) be the transposition parameter, \( m \) be the number of rows, \( n \) be the number of columns, and \( \text{lda} \) be the leading dimension. Then if

\( \text{transa} = 'N', m = 4, n = 2, \) and \( \text{lda} = 5 \), the matrix argument would be

\[
\begin{bmatrix}
(1.1, 0.11) & (1.2, 0.12) \\
(2.1, 0.21) & (2.2, 0.22) \\
(3.1, 0.31) & (3.2, 0.32) \\
(4.1, 0.41) & (4.2, 0.42)
\end{bmatrix}
\]

If \( \text{transa} = 'T', m = 4, n = 2, \) and \( \text{lda} = 5 \), the matrix argument would be

\[
\begin{bmatrix}
(1.1, 0.11) & (2.1, 0.21) & (3.1, 0.31) & (4.1, 0.41) \\
(1.2, 0.12) & (2.2, 0.22) & (3.2, 0.32) & (4.2, 0.42)
\end{bmatrix}
\]

If \( \text{transa} = 'C', m = 4, n = 2, \) and \( \text{lda} = 5 \), the matrix argument would be

\[
\begin{bmatrix}
(1.1, -0.11) & (2.1, -0.21) & (3.1, -0.31) & (4.1, -0.41) \\
(1.2, -0.12) & (2.2, -0.22) & (3.2, -0.32) & (4.2, -0.42)
\end{bmatrix}
\]

Note that care should be taken when using a leading dimension value which is different from the number of rows specified in the declaration of the two-dimensional array. For example, suppose the array \( A \) above is declared as a complex 5-by-4 matrix.

Then if \( \text{transa} = 'N', m = 3, n = 4, \) and \( \text{lda} = 4 \), the matrix argument will be

\[
\begin{bmatrix}
(1.1, 0.11) & (5.1, 0.51) & (4.2, 0.42) & (3.3, 0.33) \\
(2.1, 0.21) & (1.2, 0.12) & (5.2, 0.52) & (4.3, 0.43) \\
(3.1, 0.31) & (2.2, 0.22) & (1.3, 0.13) & (5.3, 0.53)
\end{bmatrix}
\]
Rectangular Full Packed storage allows you to store symmetric, Hermitian, or triangular matrices as compact as the Packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels. To store an \( n \)-by-\( n \) triangle (and suppose for simplicity that \( n \) is even), you partition the triangle into three parts: two \( n/2 \)-by-\( n/2 \) triangles and an \( n/2 \)-by-\( n/2 \) square, then pack this as an \( n \)-by-\( n/2 \) rectangle (or \( n/2 \)-by-\( n \) rectangle), by transposing (or transpose-conjugating) one of the triangles and packing it next to the other triangle. Since the two triangles are stored in full storage, you can use existing efficient routines on them.

There are eight cases of RFP storage representation: when \( n \) is even or odd, the packed matrix is transposed or not, the triangular matrix is lower or upper. See below for all the eight storage schemes illustrated:

\( n \) is odd, \( A \) is lower triangular

\[
\begin{array}{cccccccc}
\text{Full format} & RFP (not transposed) & RFP (transposed) \\
\mathbf{a}_{11} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{31} & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{41} & \mathbf{a}_{42} & \mathbf{a}_{43} & \mathbf{a}_{44} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{51} & \mathbf{a}_{52} & \mathbf{a}_{53} & \mathbf{a}_{54} & \mathbf{a}_{55} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{61} & \mathbf{a}_{62} & \mathbf{a}_{63} & \mathbf{a}_{64} & \mathbf{a}_{65} & \mathbf{a}_{66} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{71} & \mathbf{a}_{72} & \mathbf{a}_{73} & \mathbf{a}_{74} & \mathbf{a}_{75} & \mathbf{a}_{76} & \mathbf{a}_{77} & \mathbf{x} & \mathbf{x} \\
\end{array}
\]

\( n \) is even, \( A \) is lower triangular

\[
\begin{array}{cccccccc}
\text{Full format} & RFP (not transposed) & RFP (transposed) \\
\mathbf{a}_{11} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{21} & \mathbf{a}_{22} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{31} & \mathbf{a}_{32} & \mathbf{a}_{33} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{41} & \mathbf{a}_{42} & \mathbf{a}_{43} & \mathbf{a}_{44} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{51} & \mathbf{a}_{52} & \mathbf{a}_{53} & \mathbf{a}_{54} & \mathbf{a}_{55} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{61} & \mathbf{a}_{62} & \mathbf{a}_{63} & \mathbf{a}_{64} & \mathbf{a}_{65} & \mathbf{a}_{66} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\
\mathbf{a}_{71} & \mathbf{a}_{72} & \mathbf{a}_{73} & \mathbf{a}_{74} & \mathbf{a}_{75} & \mathbf{a}_{76} & \mathbf{a}_{77} & \mathbf{x} & \mathbf{x} \\
\end{array}
\]

\( n \) is odd, \( A \) is upper triangular

\[
\begin{array}{cccccccc}
\text{Full format} & RFP (not transposed) & RFP (transposed) \\
\mathbf{x} & \mathbf{a}_{11} & \mathbf{a}_{12} & \mathbf{a}_{13} & \mathbf{a}_{14} & \mathbf{a}_{15} & \mathbf{a}_{16} & \mathbf{a}_{17} \\
\mathbf{x} & \mathbf{a}_{22} & \mathbf{a}_{23} & \mathbf{a}_{24} & \mathbf{a}_{25} & \mathbf{a}_{26} & \mathbf{a}_{27} & \mathbf{a}_{28} \\
\mathbf{x} & \mathbf{x} & \mathbf{a}_{33} & \mathbf{a}_{34} & \mathbf{a}_{35} & \mathbf{a}_{36} & \mathbf{a}_{37} & \mathbf{a}_{38} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{44} & \mathbf{a}_{45} & \mathbf{a}_{46} & \mathbf{a}_{47} & \mathbf{a}_{48} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{55} & \mathbf{a}_{56} & \mathbf{a}_{57} & \mathbf{a}_{58} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{66} & \mathbf{a}_{67} & \mathbf{a}_{68} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{77} & \mathbf{a}_{78} \\
\end{array}
\]

\( n \) is even, \( A \) is upper triangular

\[
\begin{array}{cccccccc}
\text{Full format} & RFP (not transposed) & RFP (transposed) \\
\mathbf{x} & \mathbf{a}_{11} & \mathbf{a}_{12} & \mathbf{a}_{13} & \mathbf{a}_{14} & \mathbf{a}_{15} & \mathbf{a}_{16} & \mathbf{a}_{17} \\
\mathbf{x} & \mathbf{a}_{22} & \mathbf{a}_{23} & \mathbf{a}_{24} & \mathbf{a}_{25} & \mathbf{a}_{26} & \mathbf{a}_{27} & \mathbf{a}_{28} \\
\mathbf{x} & \mathbf{x} & \mathbf{a}_{33} & \mathbf{a}_{34} & \mathbf{a}_{35} & \mathbf{a}_{36} & \mathbf{a}_{37} & \mathbf{a}_{38} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{44} & \mathbf{a}_{45} & \mathbf{a}_{46} & \mathbf{a}_{47} & \mathbf{a}_{48} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{55} & \mathbf{a}_{56} & \mathbf{a}_{57} & \mathbf{a}_{58} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{66} & \mathbf{a}_{67} & \mathbf{a}_{68} \\
\mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{a}_{77} & \mathbf{a}_{78} \\
\end{array}
\]

Intel® MKL provides a number of routines such as \(?hfrk\), \(?sfrk\) performing BLAS operations working directly on RFP matrices, as well as some conversion routines, for instance, \(?tpttf\) goes from the standard packed format to RFP and \(?trttf\) goes from the full format to RFP.

Please refer to the Netlib site for more information.

Note that in the descriptions of LAPACK routines, arrays with RFP matrices have names ending in \( \text{fp} \).
Intel® Math Kernel Library (Intel® MKL) offers FFTW2 and FFTW3 interfaces to Intel® MKL Fast Fourier Transform and Trigonometric Transform functionality. The purpose of these interfaces is to enable applications using FFTW (www.fftw.org) to gain performance with Intel® MKL without changing the program source code.

Both FFTW2 and FFTW3 interfaces are provided in open source as FFTW wrappers to Intel® MKL. For ease of use, FFTW3 interface is also integrated in Intel® MKL.

FFT2 Notational Conventions
This appendix typically employs path notations for Windows* OS.

FFT2 Interface to Intel® Math Kernel Library
This section describes a collection of C and Fortran wrappers providing FFTW 2.x interface to Intel® MKL. The wrappers translate calls to FFTW 2.x functions into the calls of the Intel® MKL Fast Fourier Transform interface (FFT interface).

Note that Intel® MKL FFT interface operates on both single- and double-precision floating-point data types. Because of differences between FFTW and Intel® MKL FFT functionalities, there are restrictions on using wrappers instead of the FFTW functions. Some FFTW functions have empty wrappers. However, many typical FFTs can be computed using these wrappers.

Refer to chapter 11 "Fourier Transform Functions", for better understanding the effects from the use of the wrappers.

Wrappers Reference
The section provides a brief reference for the FFTW 2.x C interface. For details please refer to the original FFTW 2.x documentation available at www.fftw.org.

Each FFTW function has its own wrapper. Some of them, which are not expressly listed in this section, are empty and do nothing, but they are provided to avoid link errors and satisfy the function calls.

See Also
Limitations of the FFTW2 Interface to Intel® MKL

One-dimensional Complex-to-complex FFTs
The following functions compute a one-dimensional complex-to-complex Fast Fourier transform.

```c

.fftew_plan fftw_create_plan(int n, fftw_direction dir, int flags);

.fftew_plan fftw_create_plan_specific(int n, fftw_direction dir, int flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftw(fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist, fftw_complex *out, int ostride, int odist);

void fftw_one(fftw_plan plan, fftw_complex *in, fftw_complex *out);

void fftw_destroy_plan(fftw_plan plan);
```
Multi-dimensional Complex-to-complex FFTs

The following functions compute a multi-dimensional complex-to-complex Fast Fourier transform.

```c
fftwnd_plan fftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
fftwnd_plan fftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);
fftwnd_plan fftw3d_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);
fftwnd_plan fftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags, fftw_complex *in, int istride, fftw_complex *out, int ostride);

void fftwnd(fftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist, fftw_complex *out, int ostride, int odist);
void fftwnd_one(fftwnd_plan plan, fftw_complex *in, fftw_complex *out);
void fftwnd_destroy_plan(fftwnd_plan plan);
```

One-dimensional Real-to-half-complex/Half-complex-to-real FFTs

Half-complex representation of a conjugate-even symmetric vector of size $N$ in a real array of the same size $N$ consists of $N/2+1$ real parts of the elements of the vector followed by non-zero imaginary parts in the reverse order. Because the Intel® MKL FFT interface does not currently support this representation, all wrappers of this kind are empty and do nothing.

Nevertheless, you can perform one-dimensional real-to-complex and complex-to-real transforms using `rfftwnd` functions with `rank=1`.

See Also

Multi-dimensional Real-to-complex/complex-to-real FFTs

Multi-dimensional Real-to-complex/Complex-to-real FFTs

The following functions compute multi-dimensional real-to-complex and complex-to-real Fast Fourier transforms.

```c
rfftwnd_plan rfftwnd_create_plan(int rank, const int *n, fftw_direction dir, int flags);
rfftwnd_plan rfftw2d_create_plan(int nx, int ny, fftw_direction dir, int flags);
rfftwnd_plan rfftw3d_create_plan(int nx, int ny, int nz, fftw_direction dir, int flags);
rfftwnd_plan rfftw3d_create_plan_specific(int rank, const int *n, fftw_direction dir, int flags, fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw2d_create_plan_specific(int nx, int ny, fftw_direction dir, int flags, fftw_real *in, int istride, fftw_real *out, int ostride);
rfftwnd_plan rfftw3d_create_plan_specific(int nx, int ny, int nz, fftw_direction dir, int flags, fftw_real *in, int istride, fftw_real *out, int ostride);

void rfftwnd_real_to_complex(rfftwnd_plan plan, int howmany, fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int odist);
void rfftwnd_complex_to_real(rfftwnd_plan plan, int howmany, fftw_complex *in, int istride, int idist, fftw_real *out, int ostride, int odist);
```
void rfftwnd_one_real_to_complex(rfftwnd_plan plan, fftw_real *in, fftw_complex *out);
void rfftwnd_one_complex_to_real(rfftwnd_plan plan, fftw_complex *in, fftw_real *out);
void rfftwnd_destroy_plan(rfftwnd_plan plan);

Multi-threaded FFTW

This section discusses multi-threaded FFTW wrappers only. MPI FFTW wrappers, available only with Intel® MKL for the Linux* and Windows* operating systems, are described in section "MPI FFTW Wrappers".

Unlike the original FFTW interface, every computational function in the FFTW2 interface to Intel® MKL provides multithreaded computation by default, with the maximum number of threads permitted in FFT functions (see "Techniques to Set the Number of Threads" in Intel® MKL Developer Guide). To limit the number of threads, call the threaded FFTW computational functions:

void fftw_threads(int nthreads, fftw_plan plan, int howmany, fftw_complex *in, int istride, int idist, fftw_complex *out, int ostride, int odist);
void fftw_threads_one(int nthreads, rfftwnd_plan plan, fftw_complex *in, fftw_complex *out);
...

void rfftwnd_threads_real_to_complex( int nthreads, rfftwnd_plan plan, int howmany, fftw_real *in, int istride, int idist, fftw_complex *out, int ostride, int odist);

Compared to its non-threaded counterpart, every threaded computational function has threads_ as the second part of its name and additional first parameter nthreads. Set the nthreads parameter to the thread limit to ensure that the computation requires at most that number of threads.

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

**FFTW Support Functions**

The FFTW wrappers provide memory allocation functions to be used with FFTW:

void* fftw_malloc(size_t n);
void fftw_free(void* x);

The fftw_malloc wrapper aligns the memory on a 16-byte boundary.

If fftw_malloc fails to allocate memory, it aborts the application. To override this behavior, set a global variable fftw_malloc_hook and optionally the complementary variable fftw_free_hook:

void *(*fftw_malloc_hook) (size_t n);
void (*fftw_free_hook) (void *p);

The wrappers use the function fftw_die to abort the application in cases when a caller cannot be informed of an error otherwise (for example, in computational functions that return void). To override this behavior, set a global variable fftw_die_hook:

void (*fftw_die_hook)(const char *error_string);
void fftw_die(const char *s);
Limitations of the FFTW2 Interface to Intel® MKL

The FFTW2 wrappers implement the functionality of only those FFTW functions that Intel® MKL can reasonably support. Other functions are provided as no-operation functions, whose only purpose is to satisfy link-time symbol resolution. Specifically, no-operation functions include:

- Real-to-half-complex and respective backward transforms
- Print plan functions
- Functions for importing/exporting/forgetting wisdom
- Most of the FFTW functions not covered by the original FFTW2 documentation

Because the Intel® MKL implementation of FFTW2 wrappers does not use plan and plan node structures declared in \texttt{fftw.h}, the behavior of an application that relies on the internals of the plan structures defined in that header file is undefined.

FFTW2 wrappers define plan as a set of attributes, such as strides, used to commit the Intel® MKL FFT descriptor structure. If an FFTW2 computational function is called with attributes different from those recorded in the plan, the function attempts to adjust the attributes of the plan and recommit the descriptor. So, repeated calls of a computational function with the same plan but different strides, distances, and other parameters may be performance inefficient.

Plan creation functions disregard most planner flags passed through the \texttt{flags} parameter. These functions take into account only the following values of \texttt{flags}:

- \texttt{FFTW_IN_PLACE}
  
  If this value of \texttt{flags} is supplied, the plan is marked so that computational functions using that plan ignore the parameters related to output (\texttt{out, ostride, and odist}). Unlike the original FFTW interface, the wrappers never use the \texttt{out} parameter as a scratch space for in-place transforms.

- \texttt{FFTW_THREADSAFE}
  
  If this value of \texttt{flags} is supplied, the plan is marked read-only. An attempt to change attributes of a read-only plan aborts the application.

FFTW wrappers are generally not thread safe. Therefore, do not use the same plan in parallel user threads simultaneously.

Installing FFTW2 Interface Wrappers

Wrappers are delivered as source code, which you must compile to build the wrapper library. Then you can substitute the wrapper and Intel® MKL libraries for the FFTW library. The source code for the wrappers, makefiles, and files with lists of wrappers are located in the .\texttt{interfaces\fftw2xc\subdirectory in the Intel® MKL directory.}

Creating the Wrapper Library

Two header files are used to compile the C wrapper library: \texttt{fftw2_mkl.h} and \texttt{fftw.h}. The \texttt{fftw2_mkl.h} file is located in the .\texttt{interfaces\fftw2xc\wrappers\subdirectory in the Intel® MKL directory.}

The file \texttt{fftw.h}, used to compile libraries and located in the .\texttt{include\fftw\subdirectory in the Intel® MKL directory, slightly differs from the original FFTW (www.fftw.org) header file \texttt{fftw.h.}

The source code for the wrappers, makefiles, and files with lists of functions are located in the .\texttt{interfaces\fftw2xc\subdirectory in the Intel® MKL directory.}

A wrapper library contains wrappers for complex and real transforms in a serial and multi-threaded mode for double- or single-precision floating-point data types. A makefile parameter manages the data type. Parameters of a makefile also specify the platform (required), compiler, and data precision. The makefile comment heading provides the exact description of these parameters.

To build the library, run the \texttt{make} command on Linux* OS and macOS* or the \texttt{nmake} command on Windows* OS with appropriate parameters.
For example, on Linux OS the command

```
make libintel64
```

builds a double-precision wrapper library for Intel® 64 architecture based applications using the Intel® C++ Compiler or the Intel® Fortran Compiler (Compilers and data precision are chosen by default.)

Each makefile creates the library in the directory with Intel® MKL libraries corresponding to the platform used. For example, /lib/ia32 (on Linux OS and macOS) or \lib\ia32 (on Windows* OS).

In the names of a wrapper library, the suffix corresponds to the compiler used and the letter preceding the underscore is "c" for the C programming language.

For example,
- `fftw2xc_intel.lib` (on Windows OS);
- `libfftw2xc_intel.a` (on Linux OS and macOS);
- `fftw2xc_ms.lib` (on Windows OS);
- `libfftw2xc_gnu.a` (on Linux OS and macOS).

### Application Assembling

Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created wrapper library and the Intel® MKL library instead of the FFTW library.

### Running FFTW2 Interface Wrapper Examples

Intel® MKL provides examples to demonstrate how to use the MPI FFTW wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `./examples \fftw2xc` subdirectory in the Intel® MKL directory. To build examples, several additional files are needed: `fftw.h`, `fftw_threads.h`, `rfftw.h`, and `rfftw_threads.h`. These files are distributed with permission from FFTW and are available in `./include\fftw`. The original files can also be found in FFTW 2.1.5 at http://www.fftw.org/download.html.

An example makefile uses the `function` parameter in addition to the parameters of the corresponding wrapper library makefile (see Creating a Wrapper Library). The makefile comment heading provides the exact description of these parameters.

An example makefile normally invokes examples. However, if the appropriate wrapper library is not yet created, the makefile first builds the library the same way as the wrapper library makefile does and then proceeds to examples.

If the parameter `function=<example_name>` is defined, only the specified example runs. Otherwise, all examples from the appropriate subdirectory run. The subdirectory `./results` is created, and the results are stored there in the `<example_name>.res` files.

### Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

### MPI FFTW2 Wrappers

MPI FFTW wrappers for FFTW 2 are available only with Intel® MKL for the Linux* and Windows* operating systems.
MPI FFTW Wrappers Reference

The section provides a reference for MPI FFTW C interface.

Complex MPI FFTW

Complex One-dimensional MPI FFTW Transforms

```c
fftw_mpi_plan fftw_mpi_create_plan(MPI_Comm comm, int n, fftw_direction dir, int flags);

void fftw_mpi(fftw_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex *work);

void fftw_mpi_local_sizes(fftw_mpi_plan p, int *local_n, int *local_start, int *local_n_after_transform, int *local_start_after_transform, int *total_local_size);

void fftw_mpi_destroy_plan(fftw_mpi_plan plan);
```

Argument restrictions:

- Supported values of `flags` are `FFTW_ESTIMATE`, `FFTW_MEASURE`, `FFTW_SCRAMBLED_INPUT` and `FFTW_SCRAMBLED_OUTPUT`. The same algorithm corresponds to all these values of the `flags` parameter. If any other `flags` value is supplied, the wrapper library reports an error `’CDFT error in wrapper: unknown flags’`.
- The only supported value of `n_fields` is 1.

Complex Multi-dimensional MPI FFTW Transforms

```c
fftwnd_mpi_plan fftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction dir, int flags);

fftwnd_mpi_plan fftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz, fftw_direction dir, int flags);

fftwnd_mpi_plan fftwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction dir, int flags);

void fftwnd_mpi(fftwnd_mpi_plan p, int n_fields, fftw_complex *local_data, fftw_complex *work, fftwnd_mpi_output_order output_order);

void fftwnd_mpi_local_sizes(fftwnd_mpi_plan p, int *local_nx, int *local_x_start, int *local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);

void fftwnd_mpi_destroy_plan(fftwnd_mpi_plan plan);
```

Argument restrictions:

- Supported values of `flags` are `FFTW_ESTIMATE` and `FFTW_MEASURE`. If any other value of `flags` is supplied, the wrapper library reports an error `’CDFT error in wrapper: unknown flags’`.
- The only supported value of `n_fields` is 1.

Real MPI FFTW

Real-to-Complex MPI FFTW Transforms

```c
rfftwnd_mpi_plan rfftw2d_mpi_create_plan(MPI_Comm comm, int nx, int ny, fftw_direction dir, int flags);

rfftwnd_mpi_plan rfftw3d_mpi_create_plan(MPI_Comm comm, int nx, int ny, int nz, fftw_direction dir, int flags);
```
rfftwnd_mpi_plan rfftwnd_mpi_create_plan(MPI_Comm comm, int dim, int *n, fftw_direction dir, int flags);
void rfftwnd_mpi(rfftwnd_mpi_plan p, int n_fields, fftw_real *local_data, fftw_real *work, fftwnd_mpi_output_order output_order);
void rfftwnd_mpi_local_sizes(rfftwnd_mpi_plan p, int *local_nx, int *local_x_start, int *local_ny_after_transpose, int *local_y_start_after_transpose, int *total_local_size);
void rfftwnd_mpi_destroy_plan(rfftwnd_mpi_plan plan);

Argument restrictions:
• Supported values of flags are FFTW_ESTIMATE and FFTW_MEASURE. If any other value of flags is supplied, the wrapper library reports an error 'CDFT error in wrapper: unknown flags'.
• The only supported value of n_fields is 1.

NOTE
• Function rfftwnd_mpi_create_plan can be used for both one-dimensional and multi-dimensional transforms.
• Both values of the output_order parameter are supported: FFTW_NORMAL_ORDER and FFTW_TRANSPOSED_ORDER.

Creating MPI FFTW2 Wrapper Library
The source code for the wrappers, makefiles, and files with lists of wrappers are located in the .\interfaces\fftw2x_cdft subdirectory in the Intel® MKL directory.

A wrapper library contains C wrappers for Complex One-dimensional MPI FFTW Transforms and Complex Multi-dimensional MPI FFTW Transforms. The library also contains empty C wrappers for Real Multi-dimensional MPI FFTW Transforms. For details, see MPI FFTW Wrappers Reference.

Parameters of a makefile specify the platform (required), compiler, and data precision. Specifying the platform is required. The makefile comment heading provides the exact description of these parameters.

To build the library, run the make command on Linux* OS and macOS* or the nmake command on Windows* OS with appropriate parameters.

For example, on Linux OS the command
make libintel64
builds a double-precision wrapper library for Intel® 64 architecture based applications using Intel MPI and the Intel® C++ Compiler (compilers and data precision are chosen by default.).

A makefile creates the wrapper library in the directory with the Intel® MKL libraries corresponding to the used platform. For example,.\lib\ia32 (on Linux OS) or .\lib\ia32 (on Windows* OS).

In the wrapper library names, the suffix corresponds to the used data precision. For example, fftw2x_cdft_SINGLE.lib on Windows OS;
libfftw2x_cdft_DOUBLE.a on Linux OS.

Application Assembling with MPI FFTW Wrapper Library
Use the necessary original FFTW (www.fftw.org) header files without any modifications. Use the created MPI FFTW wrapper library and the Intel® MKL library instead of the FFTW library.
Running MPI FFTW2 Wrapper Examples

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW2. The source C code for the examples, makefiles used to run them, and files with lists of examples are located in the .\examples\fftw2x_cdft subdirectory in the Intel® MKL directory. To build examples, one additional file fftw_mpi.h is needed. This file is distributed with permission from FFTW and is available in .\include \fftw. The original file can also be found in FFTW 2.1.5 at http://www.fftw.org/download.html.

Parameters for the example makefiles are described in the makefile comment headings and are similar to the parameters of the wrapper library makefiles (see Creating MPI FFTW Wrapper Library).

The table below lists examples available in the .\examples\fftw2x_cdft\source subdirectory.

<table>
<thead>
<tr>
<th>Source file for the example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wrappers_c1d.c</td>
<td>One-dimensional Complex MPI FFTW transform, using plan = fftw_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_c2d.c</td>
<td>Two-dimensional Complex MPI FFTW transform, using plan = fftw2d_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_c3d.c</td>
<td>Three-dimensional Complex MPI FFTW transform, using plan = fftw3d_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_c4d.c</td>
<td>Four-dimensional Complex MPI FFTW transform, using plan = fftw4d_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_r1d.c</td>
<td>One-dimensional Real MPI FFTW transform, using plan = rfftw_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_r2d.c</td>
<td>Two-dimensional Real MPI FFTW transform, using plan = rfftw2d_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_r3d.c</td>
<td>Three-dimensional Real MPI FFTW transform, using plan = rfftw3d_mpi_create_plan(...)</td>
</tr>
<tr>
<td>wrappers_r4d.c</td>
<td>Four-dimensional Real MPI FFTW transform, using plan = rfftw4d_mpi_create_plan(...)</td>
</tr>
</tbody>
</table>

FFTW3 Interface to Intel® Math Kernel Library

This section describes a collection of FFTW3 wrappers to Intel® MKL. The wrappers translate calls of FFTW3 functions to the calls of the Intel® MKL Fourier transform (FFT) or Trigonometric Transform (TT) functions. The purpose of FFTW3 wrappers is to enable developers whose programs currently use the FFTW3 library to gain performance with the Intel® MKL Fourier transforms without changing the program source code.

The FFTW3 wrappers provide a limited functionality compared to the original FFTW 3.x library, because of differences between FFTW and Intel® MKL FFT and TT functionality. This section describes limitations of the FFTW3 wrappers and hints for their usage. Nevertheless, many typical FFT tasks can be performed using the FFTW3 wrappers to Intel® MKL.

The FFTW3 wrappers are integrated in Intel® MKL. The only change required to use Intel® MKL through the FFTW3 wrappers is to link your application using FFTW3 against Intel® MKL.

A reference implementation of the FFTW3 wrappers is also provided in open source. You can find it in the interfaces directory of the Intel® MKL distribution. You can use the reference implementation to create your own wrapper library (see Building Your Own Wrapper Library).

See also these resources:

Intel® MKL Release Notes for the version of the FFTW3 library supported by the wrappers.
Using FFTW3 Wrappers

The FFTW3 wrappers are a set of functions and data structures depending on one another. The wrappers are not designed to provide the interface on a function-per-function basis. Some FFTW3 wrapper functions are empty and do nothing, but they are present to avoid link errors and satisfy function calls.

This document does not list the declarations of the functions that the FFTW3 wrappers provide (you can find the declarations in the fftw3.h header file). Instead, this section comments on particular limitations of the wrappers and provides usage hints:. These are some known limitations of FFTW3 wrappers and their usage in Intel® MKL.

- The FFTW3 wrappers do not support long double precision because Intel® MKL FFT functions operate only on single- and double-precision floating-point data types (float and double, respectively). Therefore the functions with prefix fftwl_, supporting the long double data type, are not provided.
- The wrappers provide equivalent implementation for double- and single-precision functions (those with prefixes fftw_ and fftwf_, respectively). So, all these comments equally apply to the double- and single-precision functions and will refer to functions with prefix fftw_, that is, double-precision functions, for brevity.
- The FFTW3 interface that the wrappers provide is defined in the fftw3.h header file. This file is borrowed from the FFTW3.x package and distributed within Intel® MKL with permission. Additionally, the fftw3_mkl.h header file defines supporting structures and supplementary constants and macros.
- Actual functionality of the plan creation wrappers is implemented in guru64 set of functions. Basic interface, advanced interface, and guru interface plan creation functions call the guru64 interface functions. So, all types of the FFTW3 plan creation interface in the wrappers are functional.
- Plan creation functions may return a NULL plan, indicating that the functionality is not supported. So, please carefully check the result returned by plan creation functions in your application. In particular, the following problems return a NULL plan:
  - c2r and r2c problems with a split storage of complex data.
  - r2r problems with kind values FFTW_R2HC, FFTW_HC2R, and FFTW_DHT. The only supported r2r kinds are even/odd DFTs (sine/cosine transforms).
  - Multidimensional r2r transforms.
  - Transforms of multidimensional vectors. That is, the only supported values for parameter howmany_rank in guru and guru64 plan creation functions are 0 and 1.
  - Multidimensional transforms with rank > MKL_MAXRANK.
- The MKL_RODFT00 value of the kind parameter is introduced by the FFTW3 wrappers. For better performance, you are strongly encouraged to use this value rather than FFTW_RODFT00. To use this kind value, provide an extra first element equal to 0.0 for the input/output vectors. Consider the following example:

plan1 = fftw_plan_r2r_1d(n, in1, out1, FFTW_RODFT00, FFTW_ESTIMATE);
plan2 = fftw_plan_r2r_1d(n, in2, out2, MKL_RODFT00, FFTW_ESTIMATE);

Both plans perform the same transform, except that the in2/out2 arrays have one extra zero element at location 0. For example, if n=3, in1={x,y,z} and out1={u,v,w}, then in2={0,x,y,z} and out2={0,u,v,w}.
- The flags parameter in plan creation functions is always ignored. The same algorithm is used regardless of the value of this parameter. In particular, flags values FFTW_ESTIMATE, FFTW_MEASURE, etc. have no effect.
- For multithreaded plans, use normal sequence of calls to the fftw_init_threads() and fftw_plan_with_nthreads() functions (refer to FFTW documentation).
- Memory allocation function fftw_malloc returns memory aligned at a 16-byte boundary. You must free the memory with fftw_free.
The FFTW3 wrappers to Intel® MKL use the 32-bit \texttt{int} type in both LP64 and ILP64 interfaces of Intel® MKL. Use guru64 FFTW3 interfaces for 64-bit sizes.

The wrappers typically indicate a problem by returning a \texttt{NULL} plan. In a few cases, the wrappers may report a descriptive message of the problem detected. By default the reporting is turned off. To turn it on, set variable \texttt{fftw3_mkl.verbose} to a non-zero value, for example:

```c
#include "fftw3.h"
#include "fftw3_mkl.h"
fftw3_mkl.verbose = 0;
plan = fftw_plan_r2r(...);
```

The following functions are empty:
- For saving, loading, and printing plans
- For saving and loading wisdom
- For estimating arithmetic cost of the transforms.

Do not use macro \texttt{FFTW\_DLL} with the FFTW3 wrappers to Intel® MKL.

Do not use negative stride values. Though FFTW3 wrappers support negative strides in the part of advanced and guru FFTW interface, the underlying implementation does not.

Do not set a FFTW2 wrapper library before a FFT23 wrapper library or Intel® MKL in your link line application. All libraries define "fftw\_destroy\_plan" symbol and linkage in incorrect order results into expected errors.

**Building Your Own FFTW3 Interface Wrapper Library**

The FFTW3 wrappers to Intel® MKL are delivered both integrated in Intel® MKL and as source code, which can be compiled to build a standalone wrapper library with exactly the same functionality. Normally you do not need to build the wrappers yourself.

The source code for the wrappers, makefiles, and files with lists of functions are located in the .\interfaces\fftw3xc subdirectory in the Intel® MKL directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS and macOS* or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target \texttt{libia32} or \texttt{libintel64} defines the platform architecture, and the other parameters specify the compiler, size of the default integer type, and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the FFTW3 C wrappers to Intel® MKL for use from the GNU gcc* compiler on Linux OS based on Intel® 64 architecture:

```bash
cd interfaces/fftw3xc
make libintel64 compiler=gnu INSTALL_DIR=/my/path
```

This command builds the wrapper library and places the result, named \texttt{libfftw3xc\_gnu.a}, into the \texttt{/my/path} directory. The name of the resulting library is composed of the name of the compiler used and may be changed by an optional parameter \texttt{INSTALL\_LIBNAME}.

**Building an Application With FFTW3 Interface Wrappers**

Normally, the only change needed to build your application with FFTW3 wrappers replacing original FFTW library is to add Intel® MKL at the link stage (see section "Linking Your Application with Intel® Math Kernel Library" in the Intel® MKL Developer Guide).

If you recompile your application, add subdirectory \texttt{include\fftw} to the search path for header files to avoid FFTW3 version conflicts.
Sometimes, you may have to modify your application according to the following recommendations:

- The application requires
  ```
  #include "fftw3.h",
  ```
  which it probably already includes.
- The application does not require
  ```
  #include "mkl_dfti.h"
  ```
- The application does not require
  ```
  #include "fftw3_mkl.h"
  ```
  It is required only in case you want to use the MKL_RODFT00 constant.
- If the application does not check whether a NULL plan is returned by plan creation functions, this check must be added, because the FFTW3 to Intel® MKL wrappers do not provide 100% of FFTW3 functionality.

### Running FFTW3 Interface Wrapper Examples

There are some examples that demonstrate how to use the wrapper library. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3xc` subdirectory in the Intel® MKL directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. If the parameter `function=<example_name>` is defined, then only the specified example will run. Otherwise, all examples will be executed. Results of running the examples are saved in subdirectory `.\_results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

### MPI FFTW3 Wrappers

This section describes a collection of MPI FFTW wrappers to Intel® MKL.

MPI FFTW wrappers are available only with Intel® MKL for the Linux* and Windows* operating systems.

These wrappers translate calls of MPI FFTW functions to the calls of the Intel® MKL cluster Fourier transform (CFFT) functions. The purpose of the wrappers is to enable users of MPI FFTW functions improve performance of the applications without changing the program source code.

Although the MPI FFTW wrappers provide less functionality than the original FFTW3 because of differences between MPI FFTW and Intel® MKL CFFT, the wrappers cover many typical CFFT use cases.

The MPI FFTW wrappers are provided as source code. To use the wrappers, you need to build your own wrapper library (see Building Your Own Wrapper Library).

See also these resources:

- Intel® MKL Release Notes for the version of the FFTW3 library supported by the wrappers.
- www.fftw.org for a description of the MPI FFTW interface.
- Cluster FFT Functions for a description of the Intel® MKL CFFT interface.

### Building Your Own Wrapper Library

The MPI FFTW wrappers for FFTW3 are delivered as source code, which can be compiled to build a wrapper library.

The source code for the wrappers, makefiles, and files with lists of functions are located in subdirectory `.\interfaces fftw3x_cdft` in the Intel® MKL directory.

To build the wrappers,

1. Change the current directory to the wrapper directory
2. Run the `make` command on Linux* OS or the `nmake` command on Windows* OS with a required target and optionally several parameters.

The target `libia32` or `libintel64` defines the platform architecture, and the other parameters specify the compiler, size of the default `INTEGER` type, as well as the name and placement of the resulting wrapper library. You can find a detailed and up-to-date description of the parameters in the makefile.

In the following example, the `make` command is used to build the MPI FFTW wrappers to Intel® MKL for use from the GNU C compiler on Linux OS based on Intel® 64 architecture:

```bash
cd interfaces/fftw3x_cdft
make libintel64 compiler=gnu mpi=openmpi INSTALL_DIR=/my/path
```

This command builds the wrapper library using the GNU gcc compiler so that the final executable can use Open MPI, and places the result, named `libfftw3x_cdft_DOUBLE.a`, into directory `/my/path`.

**Building an Application**

Normally, the only change needed to build your application with MPI FFTW wrappers replacing original FFTW3 library is to add Intel® MKL and the wrapper library at the link stage (see section "Linking Your Application with Intel® Math Kernel Library" in the Intel® MKL Developer Guide).

When you are recompiling your application, add subdirectory `include\fftw` to the search path for header files to avoid FFTW3 version conflicts.

**Running Examples**

There are some examples that demonstrate how to use the MPI FFTW wrapper library for FFTW3. The source code for the examples, makefiles used to run them, and files with lists of examples are located in the `.\examples\fftw3x_cdft` subdirectory in the Intel® MKL directory.

Parameters of the example makefiles are similar to the parameters of the wrapper library makefiles. Example makefiles normally build and invoke the examples. Results of running the examples are saved in subdirectory `.\_results` in files with extension `.res`.

For detailed information about options for the example makefile, refer to the makefile.

**See Also**

Building Your Own Wrapper Library
This appendix presents code examples of using some Intel® MKL routines and functions. Please refer to respective chapters in the document for detailed descriptions of function parameters and operation.

**BLAS Code Examples**

**Example. Using BLAS Level 1 Function**

The following example illustrates a call to the BLAS Level 1 function `sdot`. This function performs a vector-vector operation of computing a scalar product of two single-precision real vectors `x` and `y`.

**Parameters**

- `n` Specifies the number of elements in vectors `x` and `y`.
- `incx` Specifies the increment for the elements of `x`.
- `incy` Specifies the increment for the elements of `y`.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mkl_example.h"

int main()
{
    MKL_INT  n, incx, incy, i;
    float *x, *y;
    float res;
    MKL_INT len_x, len_y;

    n = 5;
    incx = 2;
    incy = 1;

    len_x = 1+(n-1)*abs(incx);
    len_y = 1+(n-1)*abs(incy);
    x    = (float *)calloc( len_x, sizeof( float ) );
    y    = (float *)calloc( len_y, sizeof( float ) );
    if( x == NULL || y == NULL ) {
        printf( "\n Can't allocate memory for arrays\n" );
        return 1;
    }

    for (i = 0; i < n; i++) {
        x[i*abs(incx)] = 2.0;
        y[i*abs(incy)] = 1.0;
    }

    res = sdot( n, x, incx, y, incy );
    printf( "\n Result: %f\n", res );

    free( x );
    free( y );
    return 0;
}
```
As a result of this program execution, the following line is printed:

SDOT = 10.000

**Example. Using BLAS Level 1 Routine**

The following example illustrates a call to the BLAS Level 1 routine `scopy`. This routine performs a vector-vector operation of copying a single-precision real vector `x` to a vector `y`.

**Parameters**

- **n**
  Specifies the number of elements in vectors `x` and `y`.

- **incx**
  Specifies the increment for the elements of `x`.

- **incy**
  Specifies the increment for the elements of `y`.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mkl_example.h"

int main()
{
    MKL_INT  n, incx, incy, i;
    float   *x, *y;
    MKL_INT  len_x, len_y;
    n = 3;
    incx = 3;
    incy = 1;

    len_x = 10;
    len_y = 10;
    x    = (float *)calloc( len_x, sizeof( float ) );
    y    = (float *)calloc( len_y, sizeof( float ) );
    if( x == NULL || y == NULL ) {
        printf( "Can't allocate memory for arrays\n" );
        return 1;
    }
    for (i = 0; i < 10; i++) {
        x[i] = i + 1;
    }
    cblas_scopy(n, x, incx, y, incy);

    /* Print output data */
    printf("\n\n OUTPUT DATA");
}
```
As a result of this program execution, the following line is printed:

\[
Y = 1.00000 \ 4.00000 \ 7.00000
\]

**Example. Using BLAS Level 2 Routine**

The following example illustrates a call to the BLAS Level 2 routine `sger`. This routine performs a matrix-vector operation

\[
a := \alpha x^* y' + a.
\]

**Parameters**

- \(\alpha\): Specifies a scalar \(\alpha\).
- \(x\): \(m\)-element vector.
- \(y\): \(n\)-element vector.
- \(a\): \(m\times n\) matrix.

```c
#include <stdio.h>
#include <stdlib.h>
#include "mkl_example.h"

int main()
{
    MKL_INT         m, n, lda, incx, incy, i, j;
    MKL_INT         rmaxa, cmaxa;
    float           alpha;
    float          *a, *x, *y;
    CBLAS_LAYOUT    layout;
    MKL_INT         len_x, len_y;

    m = 2;
    n = 3;
    lda = 5;
    incx = 2;
    incy = 1;
    alpha = 0.5;

    len_x = 10;
    len_y = 10;
    rmaxa = m + 1;
    cmaxa = n;
    a = (float *)calloc( rmaxa*cmaxa, sizeof(float) );
    x = (float *)calloc( len_x, sizeof(float) );
    y = (float *)calloc( len_y, sizeof(float) );
    if( a == NULL || x == NULL || y == NULL ) {
        printf( "Can't allocate memory for arrays\n" );
        return 1;
    }
    // Perform the matrix-vector operation
    \[
    a := \alpha x^* y' + a.
    \]
    // Free allocated memory
    free(a);
    free(x);
    free(y);
    return 0;
}
```
if( layout == CblasRowMajor )
   lda=cmaxa;
else
   lda=rmaxa;
for (i = 0; i < 10; i++) {
   x[i] = 1.0;
   y[i] = 1.0;
}
for (i = 0; i < m; i++) {
   for (j = 0; j < n; j++) {
      a[i + j*lda] = j + 1;
   }
}
cblas_sger(layout, m, n, alpha, x, incx, y, incy, a, lda);
PrintArrayS(&layout, FULLPRINT, GENERAL_MATRIX, &m, &n, a, &lda, "A");
free(a);
free(x);
free(y);
return 0;
}

As a result of this program execution, matrix a is printed as follows:

Matrix A:

1.50000 2.50000 3.50000
1.50000 2.50000 3.50000

Example. Using BLAS Level 3 Routine

The following example illustrates a call to the BLAS Level 3 routine ssymm. This routine performs a matrix-matrix operation

\[ c := \alpha a*b' + \beta c. \]

Parameters

- \( \alpha \): Specifies a scalar.\( \alpha \).
- \( \beta \): Specifies a scalar.\( \beta \).
- \( a \): Symmetric matrix
- \( b \): \( m \)-by-\( n \) matrix
- \( c \): \( m \)-by-\( n \) matrix

#include <stdio.h>
#include <stdlib.h>
#include "mkl_example.h"

int main(int argc, char *argv[])
Code Examples

```c
{ MKL_INT m, n, i, j;
 MKL_INT lda, ldb, ldc;
 MKL_INT rmaxa, cmaxa, rmaxb, cmaxb, rmaxc, cmaxc;
 float alpha, beta;
 float *a, *b, *c;
 CBLAS_LAYOUT layout;
 CBLAS_SIDE side;
 CBLAS_UPLO uplo;
 MKL_INT ma, na, typeA;
 uplo = 'u';
 side = 'l';
 layout = CblasRowMajor;
 m = 3;
 n = 2;
 lda = 3;
 ldb = 3;
 ldc = 3;
 alpha = 0.5;
 beta = 2.0;

 if( side == CblasLeft ) {
   rmaxa = m + 1;
   cmaxa = m;
   ma = m;
   na = m;
 } else {
   rmaxa = n + 1;
   cmaxa = n;
   ma = n;
   na = n;
 }

 rmaxb = m + 1;
 cmaxb = n;
 rmaxc = m + 1;
 cmaxc = n;
 a = (float *)malloc( rmaxa*cmaxa, sizeof(float) );
 b = (float *)malloc( rmaxb*cmaxb, sizeof(float) );
 c = (float *)malloc( rmaxc*cmaxc, sizeof(float) );
 if ( a == NULL || b == NULL || c == NULL ) {
   printf("\n Can't allocate memory arrays");
   return 1;
 }

 if( layout == CblasRowMajor ) {
   lda=cmaxa;
   ldb=cmaxb;
   ldc=cmaxc;
 } else {
   lda=rmaxa;
   ldb=rmaxb;
   ldc=rmaxc;
 }

 if (uplo == CblasUpper) typeA = UPPER_MATRIX;
 else                    typeA = LOWER_MATRIX;
 for (i = 0; i < m; i++) {
   for (j = 0; j < m; j++) {
     a[i + j*lda] = 1.0;
   }
 }  
```
As a result of this program execution, matrix c is printed as follows:

Matrix C:

5.00000 5.00000
5.00000 5.00000
5.00000 5.00000

The following example illustrates a call from a C program to the Fortran version of the complex BLAS Level 1 function zdotc(). This function computes the dot product of two double-precision complex vectors.

**Example. Calling a Complex BLAS Level 1 Function from C**

In this example, the complex dot product is returned in the structure c.

```c
#include <cstdio>
#include "mkl_blas.h"
#define N 5
void main()
{
    int n, inca = 1, incb = 1, i;
    MKL_Complex16 a[N], b[N], c;
    void zdotc();
    n = N;
    for( i = 0; i < n; i++ ){
        a[i].real = (double)i; a[i].imag = (double)i * 2.0;
        b[i].real = (double)(n - i); b[i].imag = (double)i * 2.0;
    }
    zdotc( &c, &n, a, &inca, b, &incb );
    printf( "The complex dot product is: ( %6.2f, %6.2f )\n", c.real, c.imag );
}
```

**NOTE**

Instead of calling BLAS directly from C programs, you might wish to use the C interface to the Basic Linear Algebra Subprograms (CBLAS) implemented in Intel® MKL. See C Interface Conventions for more information.
Fourier Transform Functions Code Examples

This section presents code examples for functions described in the “FFT Functions” and “Cluster FFT Functions” sections in the “Fourier Transform Functions” chapter. The examples are grouped in subsections:

- Examples for FFT Functions, including Examples of Using Multi-Threading for FFT Computation
- Examples for Cluster FFT Functions
- Auxiliary data transformations.

FFT Code Examples

This section presents examples of using the FFT interface functions described in “Fourier Transform Functions” chapter.

Here are the examples of two one-dimensional computations. These examples use the default settings for all of the configuration parameters, which are specified in "Configuration Settings".

One-dimensional In-place FFT

```c
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"

float _Complex c2c_data[32];
float r2c_data[34];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle = NULL;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle = NULL;
MKL_LONG status;

/* ...put values into c2c_data[i] 0<=i<=31 */
/* ...put values into r2c_data[i] 0<=i<=31 */
status = DftiCreateDescriptor(&my_desc1_handle, DFTI_SINGLE,
                              DFTI_COMPLEX, 1, 32);
status = DftiCommitDescriptor(my_desc1_handle);
status = DftiComputeForward(my_desc1_handle, c2c_data);
/* result is c2c_data[i] 0<=i<=31 */
status = DftiFreeDescriptor(&my_desc1_handle);
status = DftiCreateDescriptor(&my_desc2_handle, DFTI_SINGLE,
                              DFTI_REAL, 1, 32);
status = DftiCommitDescriptor(my_desc2_handle);
status = DftiComputeForward(my_desc2_handle, r2c_data);
/* result is the complex value r2c_data[i] 0<=i<=31 */
/* and is stored in CCS format*/
```

One-dimensional Out-of-place FFT

```c
/* C example, float _Complex is defined in C9X */
#include "mkl_dfti.h"

float _Complex c2c_input[32];
float _Complex c2c_output[32];
float r2c_input[32];
float r2c_output[34];
DFTI_DESCRIPTOR_HANDLE my_desc1_handle = NULL;
DFTI_DESCRIPTOR_HANDLE my_desc2_handle = NULL;
MKL_LONG status;
```
Two-dimensional FFT

The following example demonstrates how you can change the default configuration settings by using the `DftiSetValue` function.

For instance, to preserve the input data after the FFT computation, the configuration of `DFTI_PLACEMENT` should be changed to "not in place" from the default choice of "in place."
The code below illustrates how this can be done:

### Changing Default Settings

```c
/* C99 example */
#include "mkl_dfti.h"

float _Complex x_in[32], x_out[32];
DFTI_DESCRIPTOR_HANDLE my_desc_handle = NULL;
MKL_LONG status;

/* ...put values into x_in[i] 0<=i<=31 */
status = DftiCreateDescriptor(&my_desc_handle, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 32);
status = DftiSetValue(my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, x_in, x_out);
status = DftiFreeDescriptor(&my_desc_handle);
/* result is the complex value x_out[i], 0<=i<=31 */
```

### Using Status Checking Functions

This example illustrates the use of status checking functions described in Chapter 11.

```c
/* C99 example */
#include "mkl_dfti.h"

DFTI_DESCRIPTOR_HANDLE desc = NULL;
MKL_LONG status;

/* ...descriptor creation and other code */
status = DftiCommitDescriptor(desc);
if (status && !DftiErrorClass(status, DFTI_NO_ERROR))
{
    printf('Error: %s\n', DftiErrorMessage(status));
}
```

### Computing 2D FFT by One-Dimensional Transforms

Below is an example where a 20-by-40 two-dimensional FFT is computed explicitly using one-dimensional transforms.

```c
/* C */
#include "mkl_dfti.h"

float _Complex data[20][40];
MKL_LONG status;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim1 = NULL;
DFTI_DESCRIPTOR_HANDLE desc_handle_dim2 = NULL;

/* ...put values into data[i][j] 0<=i<=19, 0<=j<=39 */
status = DftiCreateDescriptor(&desc_handle_dim1, DFTI_SINGLE,
    DFTI_COMPLEX, 1, 20);
status = DftiCreateDescriptor(&desc_handle_dim2, DFTI_SINGLE,
```
The following code illustrates real multi-dimensional transforms with CCE format storage of conjugate-even complex matrix. Example "Three-Dimensional REAL FFT (C Interface)" is three-dimensional out-of-place transform in C interface.

**Three-Dimensional REAL FFT**

```c
/* C99 example */
#include "mkl_dfti.h"

float input[32][100][19];
float _Complex output[32][100][10];  /* 10 = 19/2 + 1 */
DFTI_DESCRIPTOR_HANDLE my_desc_handle = NULL;
MKL_LONG status;
MKL_LONG dim_sizes[3] = {32, 100, 19};
MKL_LONG strides_out[4] = {0, 10*10*1, 10*1, 1};

//...put values into input[i][j][k] 0<=i<=31; 0<=j<=99, 0<=k<=9

status = DftiCreateDescriptor(&my_desc_handle,
                              DFTI_SINGLE, DFTI_REAL, 3, dim_sizes);
status = DftiSetValue(my_desc_handle,
                      DFTI_CONJUGATE_EVEN_STORAGE, DFTI_COMPLEX_COMPLEX);
status = DftiSetValue(my_desc_handle, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
status = DftiCommitDescriptor(my_desc_handle);
status = DftiComputeForward(my_desc_handle, input, output);
status = DftiFreeDescriptor(&my_desc_handle);

/* result is the complex output[i][j][k] 0<=i<=31; 0<=j<=99, 0<=k<=9
   and is stored in CCE format. */
```
Examples of Using OpenMP* Threading for FFT Computation

The following sample program shows how to employ internal OpenMP* threading in Intel® MKL for FFT computation.

To specify the number of threads inside Intel® MKL, use the following settings:

```plaintext
set MKL_NUM_THREADS = 1 for one-threaded mode;
set MKL_NUM_THREADS = 4 for multi-threaded mode.
```

Using Intel® MKL Internal Threading Mode (C Example)

```c
/* C99 example */
#include "mkl_dfti.h"
float data[200][100];
DFTI_DESCRIPTOR_HANDLE fft = NULL;
MKL_LONG dim_sizes[2] = {200, 100};
/* ...put values into data[i][j] 0<=i<=199, 0<=j<=99 */
DftiCreateDescriptor(&fft, DFTI_SINGLE, DFTI_REAL, 2, dim_sizes);
DftiCommitDescriptor(fft);
DftiComputeForward(fft, data);
DftiFreeDescriptor(&fft);
```

The following Example "Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region" and Example "Using Parallel Mode with Multiple Descriptors Initialized in One Thread" illustrate a parallel customer program with each descriptor instance used only in a single thread.

Specify the number of OpenMP threads for Example "Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region" like this:

```plaintext
set MKL_NUM_THREADS = 1 for Intel® MKL to work in the single-threaded mode (recommended);
set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.
```

Using Parallel Mode with Multiple Descriptors Initialized in a Parallel Region

Note that in this example, the program can be transformed to become single-threaded at the customer level but using parallel mode within Intel® MKL. To achieve this, you need to set the parameter DFTI_NUMBER_OF_TRANSFORMS = 4 and to set the corresponding parameter DFTI_INPUT_DISTANCE = 5000.

```c
/* C99 example */
#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])

// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG dim_sizes[2] = {
    ARRAY_LEN(data[0]),
    ARRAY_LEN(data[0][0])
}; /* {50, 100} */
int th;
/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */
```
// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(dim_sizes, data)
for (th = 0; th < nth; ++th)
{
    DFTI_DESCRIPTOR_HANDLE myFFT = NULL;

    DftiCreateDescriptor(&myFFT, DFTI_SINGLE, DFTI_COMPLEX, 2, dim_sizes);
    DftiCommitDescriptor(myFFT);
    DftiComputeForward(myFFT, data[th]);
    DftiFreeDescriptor(&myFFT);
}

Specify the number of OpenMP threads for Example "Using Parallel Mode with Multiple Descriptors Initialized in One Thread" like this:

set MKL_NUM_THREADS = 1 for Intel® MKL to work in the single-threaded mode (obligatory);
set OMP_NUM_THREADS = 4 for the customer program to work in the multi-threaded mode.

Using Parallel Mode with Multiple Descriptors Initialized in One Thread

/* C99 example */
#include "mkl_dfti.h"
#include <omp.h>#
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])

// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG dim_sizes[2] = {
    ARRAY_LEN(data[0]),
    ARRAY_LEN(data[0][0])
}; /* {50, 100} */
DFTI_DESCRIPTOR_HANDLE FFT[ARRAY_LEN(data)];
int th;

/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */

for (th = 0; th < nth; ++th)
    DftiCreateDescriptor(&FFT[th], DFTI_SINGLE, DFTI_COMPLEX, 2, dim_sizes);
for (th = 0; th < nth; ++th)
    DftiCommitDescriptor(FFT[th]);

// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, data)
for (th = 0; th < nth; ++th)
    DftiComputeForward(FFT[th], data[th]);
for (th = 0; th < nth; ++th)
    DftiFreeDescriptor(&FFT[th]);

The following Example "Using Parallel Mode with a Common Descriptor" illustrates a parallel customer program with a common descriptor used in several threads.

Using Parallel Mode with a Common Descriptor

#include "mkl_dfti.h"
#include <omp.h>
#define ARRAY_LEN(a) sizeof(a)/sizeof(a[0])
// 4 OMP threads, each does 2D FFT 50x100 points
MKL_Complex8 data[4][50][100];
int nth = ARRAY_LEN(data);
MKL_LONG len[2] = {ARRAY_LEN(data[0]), ARRAY_LEN(data[0][0])};
DFTI_DESCRIPTOR_HANDLE FFT;
int th;

/* ...put values into data[i][j][k] 0<=i<=3, 0<=j<=49, 0<=k<=99 */
DftiCreateDescriptor(&FFT, DFTI_SINGLE, DFTI_COMPLEX, 2, len);
DftiCommitDescriptor(FFT);

// assume data is initialized and do 2D FFTs
#pragma omp parallel for shared(FFT, data)
for (th = 0; th < nth; ++th)
    DftiComputeForward(FFT, data[th]);
DftiFreeDescriptor(&FFT);

Examples for Cluster FFT Functions

The following C example computes a 2-dimensional out-of-place FFT using the cluster FFT interface:

2D Out-of-place Cluster FFT Computation

/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;
MKL_LONG v, i, j, n, s;
Complex *in, *out;
MKL_LONG dim_sizes[2] = {nx, ny};

MPI_Init(...);

/* Create descriptor for 2D FFT */
DfticreateDescriptorDM(MPI_COMM_WORLD,
    &desc, DFTI_DOUBLE, DFTI_COMPLEX, 2, dim_sizes);
/* Ask necessary length of in and out arrays and allocate memory */
DftiGetValueDM(desc,CDFT_LOCAL_SIZE,&v);
in = (Complex*) malloc(v*sizeof(Complex));
out = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Current process performs n rows,
0 row of in corresponds to s row of virtual global array */
DftiGetValueDM(desc,CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc,CDFT_LOCAL_X_START, &s);
/* Virtual global array globalIN is defined by function f as
globalIN[i*ny+j]=f(i,j) */
for(i = 0; i < n; ++i)
    for(j = 0; j < ny; ++j) in[i*ny+j] = f(i+s,j);
/* Set that we want out-of-place transform (default is DFTI_INPLACE) */
DftiSetValueDM(desc, DFTI_PLACEMENT, DFTI_NOT_INPLACE);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc, in, out);
/* Virtual global array globalOUT is defined by function g as
globalOUT[i*ny+j]=g(i,j) Now out contains result of FFT. out[i*ny+j]=g(i+s,j) */
1D In-place Cluster FFT Computations

The C example below illustrates one-dimensional in-place cluster FFT computations effected with a user-defined workspace:

```c
/* C99 example */
#include "mpi.h"
#include "mkl_cdft.h"

DFTI_DESCRIPTOR_DM_HANDLE desc = NULL;
MKL_LONG N, v, i, n_out, s_out;
Complex *in, *work;

MPI_Init(...);
/* Create descriptor for 1D FFT */
DftiCreateDescriptorDM(MPI_COMM_WORLD, &desc, DFTI_DOUBLE, DFTI_COMPLEX, 1, N);
/* Ask necessary length of array and workspace and allocate memory */
DftiGetValueDM(desc, CDFT_LOCAL_SIZE, &v);
in = (Complex*) malloc(v*sizeof(Complex));
work = (Complex*) malloc(v*sizeof(Complex));
/* Fill local array with initial data. Local array has n elements,
0 element of in corresponds to s element of virtual global array */
DftiGetValueDM(desc, CDFT_LOCAL_NX, &n);
DftiGetValueDM(desc, CDFT_LOCAL_X_START, &s);
/* Set work array as a workspace */
DftiSetValueDM(desc, CDFT_WORKSPACE, work);
/* Virtual global array globalIN is defined by function f as globalIN[i]=f(i) */
for(i = 0; i < n; ++i) in[i] = f(i+s);
/* Commit descriptor, calculate FFT, free descriptor */
DftiCommitDescriptorDM(desc);
DftiComputeForwardDM(desc, in);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_NX, &n_out);
DftiGetValueDM(desc, CDFT_LOCAL_OUT_X_START, &s_out);
/* Virtual global array globalOUT is defined by function g as globalOUT[i]=g(i) */
Now in contains result of FFT. Local array has n_out elements,
0 element of in corresponds to s_out element of virtual global array.
in[i] == g(i+s_out) */
DftiFreeDescriptorDM(&desc);
free(in);
free(work);
MPI_Finalize();
```

Auxiliary Data Transformations

This section presents C examples for conversion from the Cartesian to polar representation of complex data and vice versa.
Conversion from Cartesian to polar representation of complex data

// Cartesian->polar conversion of complex data
// Cartesian representation: z = re + I*im
// Polar representation: z = r * exp( I*phi )
#include <mkl_vml.h>

void
variant1_Cartesian2Polar(int n,const double *re,const double *im,
                           double *r,double *phi)
{
    vdHypot(n,re,im,r);          // compute radii r[]
    vdAtan2(n,im,re,phi);       // compute phases phi[]
}

void
variant2_Cartesian2Polar(int n,const MKL_Complex16 *z,double *r,double *phi,
                           double *temp_re,double *temp_im)
{
    vzAbs(n,z,r);               // compute radii r[]
    vdPackI(n, (double*)z + 0, 2, temp_re);
    vdPackI(n, (double*)z + 1, 2, temp_im);
    vdAtan2(n,temp_im,temp_re,phi); // compute phases phi[]
}

Conversion from polar to Cartesian representation of complex data

// Polar->Cartesian conversion of complex data.
// Polar representation: z = r * exp( I*phi )
// Cartesian representation: z = re + I*im
#include <mkl_vml.h>

void
variant1_Polar2Cartesian(int n,const double *r,const double *phi,
                           double *re,double *im)
{
    vdSinCos(n,phi,im,re);      // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,re,re);           // scale real part
    vdMul(n,r,im,im);           // scale imaginary part
}

void
variant2_Polar2Cartesian(int n,const double *r,const double *phi,
                           MKL_Complex16 *z,
                           double *temp_re,double *temp_im)
{
    vdSinCos(n,phi,temp_im,temp_re); // compute direction, i.e. z[]/abs(z[])
    vdMul(n,r,temp_im,temp_im);     // scale imaginary part
    vdMul(n,r,temp_re,temp_re);     // scale real part
    vdUnpackI(n,temp_re,(double*)z + 0, 2); // fill in result.re
    vdUnpackI(n,temp_im,(double*)z + 1, 2); // fill in result.im
}
Bibliography

For more information about the BLAS, Sparse BLAS, LAPACK, ScaLAPACK, Sparse Solver, Extended Eigensolver, VM, VS, FFT, and Non-Linear Optimization Solvers functionality, refer to the following publications:

- **BLAS Level 1**

- **BLAS Level 2**

- **BLAS Level 3**

- **Sparse BLAS**


- **LAPACK**


ScaLAPACK


Sparse Solver


**Extended Eigensolver**


**VS**


**Optimization Solvers**


**Data Fitting Functions**


For a reference implementation of BLAS, sparse BLAS, LAPACK, and ScaLAPACK packages visit [www.netlib.org](http://www.netlib.org).
Glossary

$A^H$ Denotes the conjugate transpose of a general matrix $A$. See also conjugate matrix.

$A^T$ Denotes the transpose of a general matrix $A$. See also transpose.

**band matrix** A general $m$-by-$n$ matrix $A$ such that $a_{ij} = 0$ for $|i - j| > l$, where $1 < l < \min(m, n)$. For example, any tridiagonal matrix is a band matrix.

**band storage** A special storage scheme for band matrices. A matrix is stored in a two-dimensional array: columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array.

**BLAS** Abbreviation for Basic Linear Algebra Subprograms. These subprograms implement vector, matrix-vector, and matrix-matrix operations.

**BRNG** Abbreviation for Basic Random Number Generator. Basic random number generators are pseudorandom number generators imitating i.i.d. random number sequences of uniform distribution. Distributions other than uniform are generated by applying different transformation techniques to the sequences of random numbers of uniform distribution.

**BRNG registration** Standardized mechanism that allows a user to include a user-designed BRNG into the VSL and use it along with the predefined VSL basic generators.

**Bunch-Kaufman factorization** Representation of a real symmetric or complex Hermitian matrix $A$ in the form $A = PUDU^H$ (or $A = PLDL^H$) where $P$ is a permutation matrix, $U$ and $L$ are upper and lower triangular matrices with unit diagonal, and $D$ is a Hermitian block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks. $U$ and $L$ have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of $D$.

$c$ When found as the first letter of routine names, $c$ indicates the usage of single-precision complex data type.

**CBLAS** C interface to the BLAS. See BLAS.

**CDF** Cumulative Distribution Function. The function that determines probability distribution for univariate or multivariate random variable $X$. For univariate distribution the cumulative distribution function is the function of real argument $x$, which for every $x$ takes a value equal to probability of the event $A$: $X \leq x$. For multivariate distribution the cumulative distribution function is the function of a real vector $x = (x_1, x_2, \ldots, x_n)$, which, for every $x$, takes a value equal to probability of the event $A = (X_1 \leq x_1 \& X_2 \leq x_2, \& \ldots, \& X_n \leq x_n)$.

**Cholesky factorization** Representation of a symmetric positive-definite or, for complex data, Hermitian positive-definite matrix $A$ in the form $A = U^H U$ or $A = LL^H$, where $L$ is a lower triangular matrix and $U$ is an upper triangular matrix.
condition number
The number $\kappa(A)$ defined for a given square matrix $A$ as follows: $\kappa(A) = \|A\| \|A^{-1}\|$. 

conjugate matrix
The matrix $A^H$ defined for a given general matrix $A$ as follows: $(A^H)_{ij} = (a_{ji})^*$. 

conjugate number
The conjugate of a complex number $z = a + bi$ is $z^* = a - bi$. 

d
When found as the first letter of routine names, d indicates the usage of double-precision real data type. 

dot product
The number denoted $x \cdot y$ and defined for given vectors $x$ and $y$ as follows: $x \cdot y = \sum x_i y_i$. Here $x_i$ and $y_i$ stand for the $i$-th elements of $x$ and $y$, respectively. 

double precision
A floating-point data type. On Intel® processors, this data type allows you to store real numbers $x$ such that $2.23 \times 10^{-308} < |x| < 1.79 \times 10^{308}$. For this data type, the machine precision $\varepsilon$ is approximately $10^{-15}$, which means that double-precision numbers usually contain no more than 15 significant decimal digits. For more information, refer to Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture. 

eigenvalue
See eigenvalue problem. 

eigenvalue problem
A problem of finding non-zero vectors $x$ and numbers $\lambda$ (for a given square matrix $A$) such that $Ax = \lambda x$. Here the numbers $\lambda$ are called the eigenvalues of the matrix $A$ and the vectors $x$ are called the eigenvectors of the matrix $A$. 

eigenvector
See eigenvalue problem. 

elementary reflector (Householder matrix)
Matrix of a general form $H = I - \tau vv^T$, where $v$ is a column vector and $\tau$ is a scalar. In LAPACK elementary reflectors are used, for example, to represent the matrix $Q$ in the QR factorization (the matrix $Q$ is represented as a product of elementary reflectors). 

factorization
Representation of a matrix as a product of matrices. See also Bunch-Kaufman factorization, Cholesky factorization, LU factorization, LQ factorization, QR factorization, Schur factorization. 

FFTs
Abbreviation for Fast Fourier Transforms. See Chapter 11 of this book. 

full storage
A storage scheme allowing you to store matrices of any kind. A matrix $A$ is stored in a two-dimensional array $a$, with the matrix element $a_{ij}$ stored in the array element $a(i,j)$. 

Hermitian matrix
A square matrix $A$ that is equal to its conjugate matrix $A^H$. The conjugate $A^H$ is defined as follows: $(A^H)_{ij} = (a_{ji})^*$. 

I
See identity matrix. 

identity matrix
A square matrix $I$ whose diagonal elements are 1, and off-diagonal elements are 0. For any matrix $A$, $AI = A$ and $IA = A$. 

i.i.d.
Independent Identically Distributed. 

in-place
Qualifier of an operation. A function that performs its operation in-place takes its input from an array and returns its output to the same array. 

Intel® MKL
Abbreviation for Intel® Math Kernel Library.
inverse matrix

The matrix denoted as $A^{-1}$ and defined for a given square matrix $A$ as follows:

$$AA^{-1} = A^{-1}A = I,$$

$A^{-1}$ does not exist for singular matrices $A$.

$LQ$ factorization

Representation of an $m$-by-$n$ matrix $A$ as $A = LQ$ or $A = (L_0 Q)$. Here $Q$ is an $n$-by-$n$ orthogonal (unitary) matrix. For $m \leq n$, $L$ is an $m$-by-$m$ lower triangular matrix with real diagonal elements; for $m > n$, $L = \begin{bmatrix} L_1 \\ L_2 \end{bmatrix}$

where $L_1$ is an $n$-by-$n$ lower triangular matrix, and $L_2$ is a rectangular matrix.

$LU$ factorization

Representation of a general $m$-by-$n$ matrix $A$ as $A = PLU$, where $P$ is a permutation matrix, $L$ is lower triangular with unit diagonal elements (lower trapezoidal if $m > n$) and $U$ is upper triangular (upper trapezoidal if $m < n$).

machine precision

The number $\varepsilon$ determining the precision of the machine representation of real numbers. For Intel® architecture, the machine precision is approximately $10^{-7}$ for single-precision data, and approximately $10^{-15}$ for double-precision data. The precision also determines the number of significant decimal digits in the machine representation of real numbers. See also double precision and single precision.

MPI

Message Passing Interface. This standard defines the user interface and functionality for a wide range of message-passing capabilities in parallel computing.

MPICH

A freely available, portable implementation of MPI standard for message-passing libraries.

orthogonal matrix

A real square matrix $A$ whose transpose and inverse are equal, that is, $A^T = A^{-1}$, and therefore $AA^T = A^TA = I$. All eigenvalues of an orthogonal matrix have the absolute value 1.

packed storage

A storage scheme allowing you to store symmetric, Hermitian, or triangular matrices more compactly. The upper or lower triangle of a matrix is packed by columns in a one-dimensional array.

PDF

Probability Density Function. The function that determines probability distribution for univariate or multivariate continuous random variable $X$. The probability density function $f(x)$ is closely related with the cumulative distribution function $F(x)$.

For univariate distribution the relation is

$$F(x) = \int_{-\infty}^{x} f(t)dt.$$  

For multivariate distribution the relation is

$$F(x_1, x_2, \ldots, x_n) = \int_{-\infty}^{x_1} \int_{-\infty}^{x_2} \cdots \int_{-\infty}^{x_n} f(t_1, t_2, \ldots, t_n)dt_1 dt_2 \cdots dt_n.$$  

positive-definite matrix

A square matrix $A$ such that $Ax \cdot x > 0$ for any non-zero vector $x$. Here $\cdot$ denotes the dot product.

pseudorandom number generator

A completely deterministic algorithm that imitates truly random sequences.
**QR factorization**

Representation of an \( m \)-by-\( n \) matrix \( A \) as \( A = QR \), where \( Q \) is an \( m \)-by-\( m \) orthogonal (unitary) matrix, and \( R \) is \( n \)-by-\( n \) upper triangular with real diagonal elements (if \( m \geq n \)) or trapezoidal (if \( m < n \)) matrix.

**random stream**

An abstract source of independent identically distributed random numbers of uniform distribution. In this manual a random stream points to a structure that uniquely defines a random number sequence generated by a basic generator associated with a given random stream.

**RNG**

Abbreviation for Random Number Generator. In this manual the term "random number generators" stands for pseudorandom number generators, that is, generators based on completely deterministic algorithms imitating truly random sequences.

**Rectangular Full Packed (RFP) storage**

A storage scheme combining the full and packed storage schemes for the upper or lower triangle of the matrix. This combination enables using half of the full storage as packed storage while maintaining efficiency by using Level 3 BLAS/LAPACK kernels as the full storage.

**s**

When found as the first letter of routine names, \( s \) indicates the usage of single-precision real data type.

**ScaLAPACK**

Stands for Scalable Linear Algebra PACKage.

**Schur factorization**

Representation of a square matrix \( A \) in the form \( A = ZTZ^H \). Here \( T \) is an upper quasi-triangular matrix (for complex \( A \), triangular matrix) called the Schur form of \( A \); the matrix \( Z \) is orthogonal (for complex \( A \), unitary). Columns of \( Z \) are called Schur vectors.

**single precision**

A floating-point data type. On Intel\(^\circledR\) processors, this data type allows you to store real numbers \( x \) such that \( 1.18 \times 10^{-38} < |x| < 3.40 \times 10^{38} \). For this data type, the machine precision (\( \epsilon \)) is approximately \( 10^{-7} \), which means that single-precision numbers usually contain no more than 7 significant decimal digits. For more information, refer to *Intel\(^\circledR\) 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture*.

**singular matrix**

A matrix whose determinant is zero. If \( A \) is a singular matrix, the inverse \( A^{-1} \) does not exist, and the system of equations \( Ax = b \) does not have a unique solution (that is, there exist no solutions or an infinite number of solutions).

**singular value**

The numbers defined for a given general matrix \( A \) as the eigenvalues of the matrix \( A^H A \). See also SVD.

**SMP**

Abbreviation for Symmetric MultiProcessing. Intel\(^\circledR\) MKL offers performance gains through parallelism provided by the SMP feature.

**sparse BLAS**

Routines performing basic vector operations on sparse vectors. Sparse BLAS routines take advantage of vectors’ sparsity: they allow you to store only non-zero elements of vectors. See BLAS.

**sparse vectors**

Vectors in which most of the components are zeros.

**storage scheme**

The way of storing matrices. See full storage, packed storage, and band storage.

**SVD**

Abbreviation for Singular Value Decomposition. See also Singular value decomposition section in Chapter 5.

**symmetric matrix**

A square matrix \( A \) such that \( a_{ij} = a_{ji} \).
transpose

The transpose of a given matrix $A$ is a matrix $A^T$ such that $(A^T)_{ij} = a_{ji}$ (rows of $A$ become columns of $A^T$, and columns of $A$ become rows of $A^T$).

diagonal matrix A matrix $A$ such that $A = (A_1, A_2)$, where $A_1$ is an upper triangular matrix, $A_2$ is a rectangular matrix.

triangular matrix A matrix $A$ is called an upper (lower) triangular matrix if all its subdiagonal elements (superdiagonal elements) are zeros. Thus, for an upper triangular matrix $a_{ij} = 0$ when $i > j$; for a lower triangular matrix $a_{ij} = 0$ when $i < j$.

tridiagonal matrix A matrix whose non-zero elements are in three diagonals only: the leading diagonal, the first subdiagonal, and the first super-diagonal.

unitary matrix A complex square matrix $A$ whose conjugate and inverse are equal, that is, $A^* = A^{-1}$, and therefore $AA^* = A^*A = I$. All eigenvalues of a unitary matrix have the absolute value 1.

VML Abbreviation for Vector Mathematical Library. See Chapter 9 of this book.

VSL Abbreviation for Vector Statistical Library. See Chapter 10 of this book.

z When found as the first letter of routine names, $z$ indicates the usage of double-precision complex data type.
Index

?_backward_trig_transform 2326
?_commit_Helmholtz_2D 2344
?_commit_Helmholtz_3D 2344
?_commit_sph_np 2353
?_commit_sph_p 2353
?_commit_trig_transform 2322
?_forward_trig_transform 2324
?_Helmholtz_2D 2347
?_Helmholtz_3D 2347
?_init_Helmholtz_2D 2341
?_init_Helmholtz_3D 2341
?_init_sph_np 2351
?_init_sph_p 2351
?_init_trig_transform 2321
?_sph_np 2355
?_sph_p 2355
?asum 45
?axpby 349
?axpy 46
?axpyi 124
?bdsdc 793
?bdsqr 791
?cabs1 57
?ConvExec 2128
?ConvExec1D 2130
?ConvExecX 2132
?ConvExecX1D 2133
?ConvNewTask 2116
?ConvNewTask1D 2118
?ConvNewTaskX 2119
?copy 46
?CorrExec 2128
?CorrExec1D 2130
?CorrExecX 2132
?CorrExecX1D 2133
?CorrNewTask 2116
?CorrNewTask1D 2118
?CorrNewTaskX 2119
?CorrNewTaskX1D 2121
?dbtf2 1724
?dbtrf 1725
?disna 833
?dot 47
?dotc 49
?dotci 126
?doti 125
?dotu 49
?dotui 127
?dtsvb 652
?dttrf 1727
?dttrfb 425
?dttrs 465
?dttrsv 1728
?feast_hbev 1885
?feast_hbgv 1887
?feast_hcsrev 1889
?feast_hcsrv 1891
?feast_heev 1882
?feast_hgsv 1883
?feast_sbev 1885
?feast_sbgv 1887
?feast_scsrev 1889
?feast_scsrv 1891
?feast_syev 1882
?feast_sygv 1883
?gamn2 2446
?gamm2 2445
?gbcon 501
?gbequ 606
?gbequb 607
?gbmv 59
?gbtrfs 531
?gbtrfsx 533
?gbsv 633
?gbsvx 635
?gbsvxx 640
?gbtrf 421
?gbtrs 461
?gebak 858
?gebal 856
?gebr2d 2456
?gebrd 778
?gebsd 2455
?gecon 499
?geeq 602
?geequb 604
?gees 994
?geesx 997
?geev 999
?geevx 1002
?gehre 848
?gejsv 1012
?gel2 1105
?gelqf 740
?gels 935
?gelct 941
?gels 939
?gelsy 937
?gemm 102
?gemm_batch 356
?gemms 353
?gemms 353
?gemms 353
?gemq2 729
?gemv 61
?geqf 748
?geqf 733
?geqf 731
?geqrf 1106
?geqrf 724
?geqrf 726
?geqrt 727
?geqrt 1108
?geqrt 1109
?ger 63
?gerc 64
?gerfs 523
?gerfsx 525
?gerfs 755
?gerf 66
<table>
<thead>
<tr>
<th>Function</th>
<th>Code</th>
<th>Function</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>gerv2d</td>
<td>2452</td>
<td>hetrf</td>
<td>446</td>
</tr>
<tr>
<td>gesdd2d</td>
<td>2451</td>
<td>hetrf_a</td>
<td>448</td>
</tr>
<tr>
<td>gesdd</td>
<td>1009</td>
<td>hetrf_rk</td>
<td>451</td>
</tr>
<tr>
<td>gsv</td>
<td>619</td>
<td>hetrf_rook</td>
<td>449</td>
</tr>
<tr>
<td>gsvd</td>
<td>1006</td>
<td>hetri</td>
<td>586</td>
</tr>
<tr>
<td>gsvj</td>
<td>1019</td>
<td>hetri2</td>
<td>589</td>
</tr>
<tr>
<td>gsvx</td>
<td>621</td>
<td>hetri2x</td>
<td>591</td>
</tr>
<tr>
<td>gesv</td>
<td>626</td>
<td>hetsrs</td>
<td>480</td>
</tr>
<tr>
<td>getf2</td>
<td>1111</td>
<td>hets__3</td>
<td>388</td>
</tr>
<tr>
<td>getri</td>
<td>579</td>
<td>hets__a</td>
<td>481</td>
</tr>
<tr>
<td>getrs</td>
<td>459</td>
<td>hets__rook</td>
<td>483</td>
</tr>
<tr>
<td>ggbak</td>
<td>888</td>
<td>hfrk</td>
<td>1199</td>
</tr>
<tr>
<td>ggbal</td>
<td>887</td>
<td>hgeqz</td>
<td>893</td>
</tr>
<tr>
<td>gges</td>
<td>1076</td>
<td>hpcon</td>
<td>517</td>
</tr>
<tr>
<td>ggesx</td>
<td>1080</td>
<td>hpev</td>
<td>965</td>
</tr>
<tr>
<td>ggev</td>
<td>1088</td>
<td>hpevd</td>
<td>968</td>
</tr>
<tr>
<td>ggevx</td>
<td>1091</td>
<td>hpevx</td>
<td>972</td>
</tr>
<tr>
<td>ggglm</td>
<td>945</td>
<td>hpgst</td>
<td>839</td>
</tr>
<tr>
<td>ggldse</td>
<td>943</td>
<td>hpgv</td>
<td>1052</td>
</tr>
<tr>
<td>gggf</td>
<td>767</td>
<td>hpgvd</td>
<td>1055</td>
</tr>
<tr>
<td>gggqrf</td>
<td>770</td>
<td>hpgvx</td>
<td>1060</td>
</tr>
<tr>
<td>ggsvd</td>
<td>1023</td>
<td>hpmv</td>
<td>74</td>
</tr>
<tr>
<td>ggsvp</td>
<td>913</td>
<td>hpr</td>
<td>75</td>
</tr>
<tr>
<td>gsum2d</td>
<td>2448</td>
<td>hpr2</td>
<td>77</td>
</tr>
<tr>
<td>gtcon</td>
<td>503</td>
<td>hprfs</td>
<td>571</td>
</tr>
<tr>
<td>gthr</td>
<td>127</td>
<td>hpsv</td>
<td>717</td>
</tr>
<tr>
<td>gthrz</td>
<td>128</td>
<td>hpsvx</td>
<td>718</td>
</tr>
<tr>
<td>gtrfs</td>
<td>539</td>
<td>hptrd</td>
<td>809</td>
</tr>
<tr>
<td>gtsv</td>
<td>647</td>
<td>hptrf</td>
<td>456</td>
</tr>
<tr>
<td>gtsvx</td>
<td>649</td>
<td>hptri</td>
<td>597</td>
</tr>
<tr>
<td>gttfr</td>
<td>424</td>
<td>hptrs</td>
<td>491</td>
</tr>
<tr>
<td>gttors</td>
<td>463</td>
<td>hsein</td>
<td>862</td>
</tr>
<tr>
<td>hbev</td>
<td>975</td>
<td>hseqr</td>
<td>859</td>
</tr>
<tr>
<td>hbevd</td>
<td>979</td>
<td>jacobi</td>
<td>2387</td>
</tr>
<tr>
<td>hbevX</td>
<td>983</td>
<td>jacobi_delete</td>
<td>2386</td>
</tr>
<tr>
<td>hbgst</td>
<td>843</td>
<td>jacobi_init</td>
<td>2384</td>
</tr>
<tr>
<td>hbgvd</td>
<td>1068</td>
<td>jacobi_solve</td>
<td>2385</td>
</tr>
<tr>
<td>hbgvd</td>
<td>1068</td>
<td>jacobix</td>
<td>2388</td>
</tr>
<tr>
<td>hbgv</td>
<td>1064</td>
<td>lacgv</td>
<td>1100</td>
</tr>
<tr>
<td>hbgvx</td>
<td>1073</td>
<td>lacn2</td>
<td>1112</td>
</tr>
<tr>
<td>htbrd</td>
<td>815</td>
<td>lacp2</td>
<td>1214</td>
</tr>
<tr>
<td>hecon</td>
<td>513</td>
<td>lacpy</td>
<td>1113</td>
</tr>
<tr>
<td>hecon__3__514</td>
<td></td>
<td>lacrm</td>
<td>1100</td>
</tr>
<tr>
<td>heequb</td>
<td>616</td>
<td>lacrt</td>
<td>1120</td>
</tr>
<tr>
<td>heev</td>
<td>949</td>
<td>lagge</td>
<td>1223</td>
</tr>
<tr>
<td>heevd</td>
<td>951</td>
<td>laghe</td>
<td>1224</td>
</tr>
<tr>
<td>heevr</td>
<td>960</td>
<td>lagsy</td>
<td>1225</td>
</tr>
<tr>
<td>heevx</td>
<td>955</td>
<td>lamch</td>
<td>1221</td>
</tr>
<tr>
<td>hegst</td>
<td>836</td>
<td>lamsh</td>
<td>1683</td>
</tr>
<tr>
<td>hegvd</td>
<td>1042</td>
<td>lange</td>
<td>1115</td>
</tr>
<tr>
<td>hegvx</td>
<td>1047</td>
<td>lanhe</td>
<td>1117</td>
</tr>
<tr>
<td>hemm</td>
<td>105</td>
<td>lansy</td>
<td>1116</td>
</tr>
<tr>
<td>hemv</td>
<td>70</td>
<td>lantr</td>
<td>1118</td>
</tr>
<tr>
<td>her</td>
<td>71</td>
<td>laplmr</td>
<td>1120</td>
</tr>
<tr>
<td>her2</td>
<td>72</td>
<td>lapmt</td>
<td>1122</td>
</tr>
<tr>
<td>her2k</td>
<td>109</td>
<td>lapy2</td>
<td>1123</td>
</tr>
<tr>
<td>herfs</td>
<td>562</td>
<td>lapy3</td>
<td>1123</td>
</tr>
<tr>
<td>herfsx</td>
<td>564</td>
<td>larcn</td>
<td>1215</td>
</tr>
<tr>
<td>herk</td>
<td>107</td>
<td>larfb</td>
<td>1124</td>
</tr>
<tr>
<td>hhes</td>
<td>698</td>
<td>larfg</td>
<td>1127</td>
</tr>
<tr>
<td>hhes__aa__699</td>
<td></td>
<td>lartf</td>
<td>1129</td>
</tr>
<tr>
<td>hhes__rk__701</td>
<td></td>
<td>larfxc</td>
<td>1131</td>
</tr>
<tr>
<td>hhesv</td>
<td>703</td>
<td>largv</td>
<td>1134</td>
</tr>
<tr>
<td>hhesvx</td>
<td>706</td>
<td>lartg</td>
<td>1140</td>
</tr>
<tr>
<td>heswapr</td>
<td>1196</td>
<td>lartgs</td>
<td>1141</td>
</tr>
<tr>
<td>hetrd</td>
<td>801</td>
<td>lascl</td>
<td>1141</td>
</tr>
</tbody>
</table>
auxiliary routines (continued)
ScalAPACK 1507

B
balancing a matrix 856
band storage scheme 2541
basic quasi-number generator
  Niederreiter 2031
  Sobol 2031
basic random number generators
  GFSR 2031
  MCG, 32-bit 2031
  MCG, 59-bit 2031
  Mersenne Twister
    MT19937 2031
    MT2203 2031
  MRG 2031
  Wichmann-Hill 2031
bdsdc 793
Bernoulli 2093
Beta 2084
bidiagonal matrix
  LAPACK 777
  ScalAPACK 1412
Binomial 2096
BLACS
  ?gamm2d 2446
  ?gama2d 2445
  ?gebr2d 2456
  ?gebs2d 2452
  ?gesd2d 2451
  ?gsum2d 2448
  ?trbr2d 2457
  ?trbs2d 2455
  ?trrv2d 2453
  ?trsd2d 2452
  blacs_abort 2466
  blacs_barrier 2468
  blacs_exit 2466
  blacs_freebuff 2465
  blacs_get 2459
  blacs_gridexit 2465
  blacs_gridinfo 2467
  blacs_gridinit 2462
  blacs_gridmap 2463
  blacs_pcoord 2468
  blacs_pinfo 2458
  blacs_pnum 2467
  blacs_setup 2459
  BLAS Code Examples 2557
  BLAS Level 1 routines
    ?asum 44, 45
    ?axpby 349
    ?axpy 44, 46
    ?cabs1 44, 57
    ?copy 44, 46
    ?dot 44, 47
    ?dotc 44, 49
    ?dotu 44, 49
    ?nrm2 44, 49
    ?rot 44, 51
    ?rotg 44, 52
    ?rotm 44, 52
    ?rotmg 53
    ?scal 44, 55
    ?sdot 44, 48
    ?swap 44, 55
    code example 2557, 2558
    ?amax 44, 56
    ?amin 44, 57
  BLAS Level 2 routines
    ?gbmv 58, 59
    ?gemv 58, 61
    ?ger 58, 63
    ?gerc 58, 64
    ?geru 58, 66
    ?hbmv 58, 67
    ?hemv 58, 70
    ?her 58, 71
    ?her2 58, 72
    ?hpmv 58, 74
    ?hpr 58, 75
    ?hpr2 58, 77
    ?sbmv 58, 78
    ?spmv 58, 81
    ?spr 58, 82
    ?spr2 58, 84
    ?symv 58, 85
    ?syrm 58, 87
    ?sysr 58, 88
    ?tbbm 58, 89
    ?tbsv 58, 92
    ?tpmv 58, 95
    ?tpsv 58, 97
    ?trmv 58, 98
    ?trsv 58, 100
    code example 2559
  BLAS Level 3 routines
    ?gemm 101, 102
    ?gemm_batch 356
    ?gemm3m 353
    ?gemm3m_batch 360
    ?hemm 101, 105
    ?her2k 101, 109
    ?herk 101, 107
    ?symm 101, 112
BLAS Level 3 routines (continued)
?syrk 101, 114
?tftsm 1200
?trmm 101, 119
?trsm 101, 121
?trsm_batch 363
code example 2560
mkl_?gemm_compact 257, 260

BLAS routines
routine groups 41
BLAS-like extensions 348
BLAS-like transposition routines
mkl_?imatcopy 366
mkl_?omatadd 371
mkl_?omatcopy 367
mkl_?omatcopy2 369
block reflector
general matrix
ScalAPACK 1603
general rectangular matrix
LAPACK 1124
ScalAPACK 1590
triangular factor
LAPACK 1129
ScalAPACK 1598, 1609
block-cyclic distribution 1234, 1235, 2243
block-splitting method 2031
BRNG 2025, 2026, 2031
Bunch-Kaufman factorization
Hermitian matrix
packed storage 456
symmetric matrix
packed storage 454

C Datatypes 39
C interface conventions
BLAS, Sparse BLAS 42
LAPACK 398
Cauchy 2074
cbcsd 928
CBLAS
arguments 43
CBLAS to the BLAS 42
cblas_cgemmt 350
cblas_dgemm_alloc 374, 375
cblas_dgemm_compute 384, 387
cblas_dgemm_free 393
cblas_dgemm_pack 377, 380
cblas_dgemmt 350
cblas_sgemm_s8u8s16 393
cblas_sgemm_alloc 374, 375
cblas_sgemm_compute 384, 387
cblas_sgemm_free 393
cblas_sgemm_pack 377, 380
cblas_zgemmt 350
cblas_zgemmt 350
cgbcn 501
cgbrfs 533
cgbsv 635
cgbtrs 461
cgecon 499
cgeqpf 731
cgtrfs 539
checon_3 514
chsv_aa 699
chesv_rk 701
chswrap 1196
chtrfs_aa 448
chtrfs_rk 451
chetri_3 594
chetri2 589
chetrix 591
chetsr_3 488
chetsr_aa 481
chetsr2 485
chgeqz 893
Cholesky 262
Cholesky factorization
Hermitian positive semidefinite matrix 430
Hermitian positive-definite matrix
band storage 434, 472, 673, 1249, 1263
packed storage 433, 668
split 844
symmetric positive semidefinite matrix 430
symmetric positive-definite matrix
band storage 434, 472, 673, 1249, 1263
packed storage 433, 668
chseqr 859
clahqr2 1681
clapmr 1120
clapmt 1122
clarf 1689
clarfb 1124
clarf 1129
clauum 1193
clubr 1689
cluster_sparse_solver 1796
cluster_sparse_solver iparm parameter 1809
cluster_sparse_solver 64 1801
CNR, support functions for
mkl_cbrw_get 2427
mkl_cbrw_get_auto_branch 2428
mkl_cbrw_set 2427
code examples
BLAS Level 1 function 2557
BLAS Level 1 routine 2558
BLAS Level 2 routine 2559
BLAS Level 3 routine 2560
communication subprograms 1234
compact 253
Compact 263, 265, 266, 268, 269, 271
complex Hermitian matrix
1-norm value
LAPACK 1117
ScalAPACK 1561
Frobenius norm
LAPACK 1117
ScalAPACK 1561
infinity- norm
LAPACK 1117
ScalAPACK 1561
largest absolute value of element
LAPACK 1117
ScalAPACK 1561
complex matrix
complex elementary reflector
ScalAPACK 1606
complex symmetric matrix
1-norm value 1116
Frobenius norm 1116
infinity- norm 1116
largest absolute value of element 1116
complex vector
1-norm using true absolute value
conjugation (continued)

1-norm using true absolute value (continued)

LAPACK 1105
ScalAPACK 1518
conjugation
LAPACK 1100
ScalAPACK 1513

complex vector conjugation
LAPACK 1100
ScalAPACK 1513

compressed sparse vectors 123
computational node 2026
Computational Routines 722
condition number

band matrix 501

general matrix
LAPACK 499
ScalAPACK 1270, 1272, 1275
Hermitian matrix
packed storage 517
Hermitian positive-definite matrix
band storage 507
packed storage 506
tridiagonal 508

symmetric matrix
packed storage 516
symmetric positive-definite matrix
band storage 507
packed storage 506
tridiagonal 508

triangular matrix
band storage 521
packed storage 520

tridiagonal matrix 503

configuration parameters, in FFT interface 2192
Configuration Settings, for Fourier transform
functions 2192, 2224
Continuous Distribution Generators 2059
Continuous Distributions 2062
ConvCopyTask 2136
ConvDeleteTask 2135

converting a sparse vector into compressed storage form
and writing zeros to the original vector 128
converting compressed sparse vectors into full storage form 130
ConvInternalPrecision 2125
Convolution and Correlation 2111

Convolution Functions

?ConvExec 2128
?ConvExecX 2132
?ConvExecX1D 2133
?ConvNewTask 2116
?ConvNewTask1D 2118
?ConvNewTaskX 2119
?ConvNewTaskX1D 2121
CorrSetDecimation 2127
CorrSetInternalPrecision 2125
Correlation Functions:CorrSetMode 2124
Correlation Functions:CorrSetStart 2126
CorrSetInternalDecimation 2127
CorrSetInternalPrecision 2125
CorrSetMode 2124
CorrSetStart 2126
cosine-sine decomposition

See also LAPACK routines, cs decomposition 928

CSD (cosine-sine decomposition)

LAPACK 928, 1032
csycon_3 511
csyconv 1101
csysv_aa 683
csysv_rk 686
csyswapr 1194
csytrf_aa 440
csytrf_rk 443
csytri_3 593
csytri2 588
csytri2x 590
csytrs_3 487
csytrs_aa 477
csytrs2 484
ctrexc 877
ctrmv 1732
cunbdbb 931
cuncsd 1032
cuncsd2by1 1035
cungbr 787

D

data type
in VM 1894
shorthand 34
Data Types 2034
Datatypes, C language 39
dbbcsd 928
dbsdc 793
dcg_check 1845
dcg_get 1847
dcg_init 1844
dcgmrhs_check 1849
dcgmrhs_get 1852
dcgmrhs_init 1848
DeleteStream 2045
desrcinit 1744
descriptor configuration
cluster FFT 2233
descriptor manipulation
cluster FFT 2233
DF task 2470
dfConstruct1D 2500
dfEditIdxPtr 2495
dfEditPPSpline1D 2485
dfEditPtr 2492
dfDeleteTask 2520
dfIntegrate1D 2509
dfIntegrateEx1D 2509
dfIntegrCallBack 2498
dfIntegrPtr 2497
dfInterpolate1D 2501
dfInterpolateEx1D 2501
dfNewTask1D 2482
dfQueryIdxPtr 2498
dfQueryPtr 2497
dfSearchCells1D 2513
dfSearchCellsEx1D 2513
dfmgres_check 1853
dfmgres_get 653
dfmgres_init 1852
dfEditVal 2493
dfQueryVal 2498
dfsConstruct1D 2500
dfsEditIdxPtr 2495
dfsEditPPSpline1D 2485
dfsEditPtr 2492
dfsIntegrate1D 2509
dfsIntegrateEx1D 2509
dfsIntegrCallBack 2516
dfsIntegrPtr 2497
dfsInterpolate1D 2501
dfsInterpolateEx1D 2501
dfsNewTask1D 2482
dfsQueryIdxPtr 2498
dfsQueryPtr 2497
dfsSearchCells1D 2513
dfsSearchCellsEx1D 2513
DFT routines
descriptor configuration
DftiSetValue 2218
DftiCommitDescriptor 2215
DftiCommitDescriptorDM 2234
DftiComputeBackward 2222
DftiComputeBackwardDM 2237
DftiComputeForward 2221
DftiComputeForwardDM 2235
DftiCopyDescriptor 2217
DftiCreateDescriptor 2214
DftiCreateDescriptorDM 2233
DftiErrorClass 2227
DftiErrorMessage 2228
DftiFreeDescriptor 2216
DftiFreeDescriptorDM 2235
DftiGetValue 2219
DftiGetValueDM 2240
DftiSetValue 2218
DftiSetValueDM 2238
dgbcon 501
dgbrfsx 533
dgsvx 635
dgbtrs 461
dgecon 499
dgejsv 1012
dgeqpf 731
dgesvj 1019
dgtrfs 539
dhseqz 893
dhseqr 859
diagonal elements
LAPACK 1167
ScaLAPACK 1614
diagonal pivoting factorization
Hermitian indefinite matrix 706
symmetric indefinite matrix 691
diagonally dominant tridiagonal matrix
solving systems of linear equations 465
diagonally dominant-like banded matrix
solving systems of linear equations 1257
diagonally dominant-like tridiagonal matrix
solving systems of linear equations 1260
dimension 2539
Direct Sparse Solver (DSS) Interface Routines 1818
Discrete Distribution Generators 2059, 2060
Discrete Distributions 2087
Discrete Fourier Transform
DftiSetValue 2218
distributed complex matrix
transposition 2309, 2310
distributed general matrix
matrix-vector product 2259, 2261
rank-1 update 2263
rank-1 update, unconjugated 2267
rank-i update, conjugated 2265
distributed Hermitian matrix
matrix-vector product 2268, 2270
rank-1 update 2272
rank-2 update 2274
rank-k update 2297
distributed matrix equation
AX = B 2314
distributed matrix-matrix operation
rank-k update
distributed Hermitian matrix 2297
transposition
complex matrix 2309
complex matrix, conjugated 2310
real matrix 2308
distributed matrix-vector operation
product
Hermitian matrix 2268, 2270
symmetric matrix 2276, 2277
triangular matrix 2283, 2285
rank-1 update
Hermitian matrix 2272
symmetric matrix 2279
rank-1 update, conjugated 2265
rank-1 update, unconjugated 2267
rank-2 update
Hermitian matrix 2274
symmetric matrix 2281
distributed real matrix
transposition 2308

distributed symmetric matrix
  matrix-vector product 2276, 2277
  rank-1 update 2279
  rank-2 update 2281

distributed triangular matrix
  matrix-vector product 2283, 2285
  solving systems of linear equations 2287

distributed vector-scalar product 2255

distributed vectors
  adding magnitudes of vector elements 2247
  copying 2250
  dot product
    complex vectors 2253
    complex vectors, conjugated 2252
    real vectors 2251
  Euclidean norm 2254
  global index of maximum element 2246
  linear combination of vectors 2248
  sum of vectors 2248

swapping 2256

vector-scalar product 2255

distributed-memory computations 1234

Distribution Generators 2058

Distribution Generators Supporting Accurate Mode 2060

driver (continued)
  expert 1237
  simple 1237

Driver Routines 618, 935
dss_create 1821
dstegr2 1712
dstegr2a 1715
dstegr2b 1719
dsysv_3 511
dsysv_rk 686
dsyswapr 1194
dsytf2 440
dsytrf_rk 443
dsytri_3 593
dsytri2 588
dsytri2x 590
dsytrs_3 487
dsytrs2 484
dtrex 877
dtrmv 1732
dtrnlspc_check 2372
dtrnlspc_delete 2376
dtrnlspc_get 2375
dtrnlspc_init 2370
dtrnlspc_solve 2373
dtrnlspc_check 2379
dtrnlspc_delete 2383
dtrnlspc_get 2382
dtrnlspc_init 2377
dtrnlspc_solve 2381
dzsum1 1105

eigenpairs, sorting 1710

eigenvalue problems
  general matrix 846, 884, 1398
  generalized form 834
  Hermitian matrix 795
  symmetric matrix 795
  symmetric tridiagonal matrix 1722, 1731

eigenvalues
  bisection 1692
  eigenvalue problems 795
  eigenvectors 1715
  eigenvalues and eigenvectors 1715
  eigenvalue problems 795
  scaled 1688

elementary reflector
  complex matrix 1606
  general matrix 1600
    general rectangular matrix
    LAPACK 1131
    ScaLAPACK 1588, 1593
    LAPACK generation 1127
    ScaLAPACK generation 1596

Enter index keyword 267, 272–274, 276, 295, 311

error diagnostics, in VM 1897

error estimation for linear equations
  distributed tridiagonal coefficient matrix 1284

error handling
  pxerbla 2406
  xerbla 41, 1897, 2243
  error handling for fatal errors 2408

errors in solutions of linear equations
  banded matrix 533
triangular matrix (continued)
  distributed tridiagonal coefficient matrix 1284
  general matrix
  band storage 531
Hermitian indefinite matrix 564
  Hermitian matrix
  packed storage 571
  Hermitian positive-definite matrix
  band storage 550
  packed storage 549
symmetric indefinite matrix 556
symmetric matrix
  packed storage 569
  symmetric positive-definite matrix
  band storage 550
  packed storage 549
triangular matrix
  band storage 576
  packed storage 574
tridiagonal matrix 539
Estimates 2499
Euclidean norm
  of a distributed vector 2254
  of a vector 50
exception handling 2404
expert driver 1237
Exponential 2069
Extended Eigensolver 1869
Extended Eigensolver interface
  ?feast_hbev 1885
  ?feast_hbgv 1887
  ?feast_hcsrev 1889
  ?feast_hcsrgv 1891
  ?feast_heev 1882
  ?feast_hegv 1883
  ?feast_hrci 1879
  ?feast_sbey 1885
  ?feast_sbgv 1887
  ?feast_scscrv 1889
  ?feast_scscrgv 1891
  ?feast_srci 1879
  ?feast_syev 1882
  ?feast_sygv 1883
feastinit 1873
Extended Eigensolvers 1871–1876, 1879, 1881–1883, 1885, 1887, 1889, 1891

F
factorization
  Bunch-Kaufman
    LAPACK 414
    ScalAPACK 1239
  Cholesky
    LAPACK 414
    ScalAPACK 1661
diagonal pivoting
  Hermitian matrix
    packed 718
  symmetric matrix
    packed 714
LU
  LAPACK 414
  ScalAPACK 1239
orthogonal
  LAPACK 722
  ScalAPACK 1296
triangular factorization 414, 1239
fast Fourier transform
  DftiCommitDescriptor 2215
  DftiCommitDescriptorDM 2234
  DftiComputeBackward 2222
  DftiComputeBackwardDM 2237
  DftiComputeForwardDM 2235
  DftiCopyDescriptor 2217
  DftiCreateDescriptor 2214
  DftiCreateDescriptorDM 2233
  DftiErrorClass 2227
  DftiErrorMessage 2228
  DftiFreeDescriptor 2216
  DftiFreeDescriptorDM 2235
  DftiGetValue 2219
  DftiGetValueDM 2240
  DftiSetValueDM 2238
fast Fourier Transform
  DftiComputeForward 2221
fatal error, handling 2408
feastinit 1873
FFT computation
  cluster FFT 2233
FFT functions
  descriptor manipulation
    DftiCommitDescriptor 2215
    DftiCommitDescriptorDM 2234
    DftiCopyDescriptor 2217
    DftiCreateDescriptor 2214
    DftiCreateDescriptorDM 2233
    DftiFreeDescriptor 2216
    DftiFreeDescriptorDM 2235
  DFT computation
    DftiComputeBackward 2222
    DftiComputeForward 2221
    FFT computation
    DftiComputeForwardDM 2235
      status checking
    DftiErrorClass 2227
    DftiErrorMessage 2228
FFT Interface 2192
FFT routines
  descriptor configuration
    DftiGetValuem 2219
    DftiGetValue 2219
    DftiGetValueDM 2240
    DftiSetValuem 2238
  FFT computation
    DftiComputeBackwardDM 2237
  FFTW interface to Intel(R) MKL
    for FFTW2 2545
    for FFTW3 2552
  fill-in, for sparse matrices 2523
  finding
    element of a vector with the largest absolute value 56
    element of a vector with the largest absolute value of the real part and its global index 1517
    element of a vector with the smallest absolute value 57
    index of the element of a vector with the largest absolute value of the real part 1513
font conventions 34
format
  mkl_get_format_compact 269
  free_Helmholtz_2D 2350
  free_Helmholtz_3D 2350
  free_sph_np 2357
  free_sph_p 2357
  free_trig_transform 2328
Frobenius norm
  complex Hermitian matrix 1117, 1561
Frobenius norm (continued)
complex symmetric matrix 1116
general rectangular matrix 1115, 1557
real symmetric matrix 1116, 1561
trapezoidal matrix 1118
triangular matrix 1118, 1563
upper Hessenberg matrix 1559
full storage scheme 2541
full-storage vectors 123
function name conventions, in VM 1895

G
Gamma 2082, 2086
gathering sparse vector's elements into compressed form and writing zeros to these elements 128
Gaussian 2064
GaussianMV 2066
gbcon 501
gbsvx 635
gbtrs 461
gecon 499
gemm 350
general distributed matrix
scalar-matrix-matrix product 2293
general matrix
block reflector 1603
copying 1746
eigenvalue problems 846, 884, 1398
elementary reflector 1600
estimating the condition number
band storage 501
inverting matrix
LAPACK 579, 580
ScalAPACK 1287
LQ factorization 740, 1310
LU factorization
band storage 421, 1241, 1243, 1724, 1725
matrix-vector product
band storage 59
multiplying by orthogonal matrix
from LQ factorization 1649
from QR factorization 1645
from RQ factorization 1652
multiplying by unitary matrix
from LQ factorization 1649
from QR factorization 1645
from RQ factorization 1652
QL factorization
LAPACK 748
ScalAPACK 1321
QR factorization
with pivoting 731, 733, 1299
rank-1 update 63
rank-1 update, conjugated 64
rank-1 update, unconjugated 66
reduction to bidiagonal form 1526
reduction to upper Hessenberg form 1529
RQ factorization
LAPACK 755
ScalAPACK 1361
scalar-matrix-matrix product 102, 353, 356, 360
Schur factorization reordering 1672
solving systems of linear equations
band storage
LAPACK 461
ScalAPACK 1255
general rectangular distributed matrix
computing scaling factors 1292
equilibration 1292
general rectangular matrix
1-norm value
LAPACK 1115
ScalAPACK 1557
block reflector
LAPACK 1124
ScalAPACK 1590
elementary reflector
LAPACK 1593
ScalAPACK 1588
Frobenius norm
LAPACK 1115
ScalAPACK 1557
infinity-norm
LAPACK 1115
ScalAPACK 1557
largest absolute value of element
LAPACK 1115
ScalAPACK 1557
LQ factorization
LAPACK 1105
ScalAPACK 1531
multiplication
LAPACK 1141
ScalAPACK 1611
QL factorization
ScaLAPACK 1533
QR factorization
LAPACK 1106
ScalAPACK 1535
reduction of first columns
ScalAPACK 1553
reduction to bidiagonal form 1541
row interchanges
LAPACK 1169
ScalAPACK 1332, 1537
scaling 1569
general square matrix
trace 1622
general triangular matrix
LU factorization
band storage 1519
general tridiagonal triangular matrix
LU factorization
band storage 1521
generalized eigenvalue problems
complex Hermitian positive-definite problem
band storage 843
packed storage 839
complex Hermitian-definite problem 1669
real symmetric-definite problem
band storage 841
packed storage 838
See also LAPACK routines, generalized eigenvalue problems 834
Generalized LLS Problems 943
Generalized Nonsymmetric Eigenproblems 1076
generalized Schur factorization 1134
Generalized Singular Value Decomposition 912
generalized Sylvester equation 906
Generalized SymmetricDefinite Eigenproblems 1037
generation methods 2026
Geometric 2094
geqpft

gsvdx

GetBrngProperties

GetNumRegBrngs

getrf2

GetStreamSize

GetStreamStateBrng

GFSR

ggsvp3

Givens rotation

modified Givens transformation parameters of sparse vectors

parameters

global array

global index of maximum element of a distributed vector

global matrix

gtrfs

Gumbel

Helmholtz problem

three-dimensional

two-dimensional

Helmholtz problem on a sphere

non-periodic

periodic

Hermitian distributed matrix

rank-n update

scalar-matrix-matrix product

Hermitian matrix

Bunch-Kaufman factorization

packed storage

eigenvalues and eigenvectors

estimating the condition number

packed storage

generalized eigenvalue problems

inverting the matrix

packed storage

matrix-vector product

band storage

packed storage

rank-1 update

rank-2 update

packed storage

rank-2k update

rank-k update

reducing to standard form

ScalAPACK

reducing to tridiagonal form

ScalAPACK

scalar-matrix-matrix product

scaling

solving systems of linear equations

packed storage

Hermitian positive definite distributed matrix

computing scaling factors

equilibration

Hermitian positive semidefinite matrix

Cholesky factorization

Hermitian positive-definite distributed matrix

inverting the matrix

Hermitian positive-definite matrix

Cholesky factorization

band storage

solving systems of linear equations (continued)

Cholesky factorization (continued)

packed storage

estimating the condition number

band storage

packed storage

inverting the matrix

packed storage

solving systems of linear equations

band storage

packed storage

Hermitian positive-definite tridiagonal matrix

Hessenberg matrix

eigenvalues

heswapr

hetri2

hetri2x

hgeqz

Householder matrix

LAPACK

ScaLAPACK

hseqr

Hypergeometric

i?amax

i?amin

i?max1

IBM ESSL library

IEEE arithmetic

IEEE standard

IEEE standard implementation

signbit position

ilaver

ILU0 preconditioner

Incomplete LU Factorization Technique

increment

NewAbstractStream

functions

mkl_set_env_mode

Interface Consideration

inverse matrix. inverting a matrix

inverting a matrix

general matrix

LAPACK

ScaLAPACK

Hermitian matrix

packed storage

Hermitian positive-definite matrix

packed storage

symmetric matrix

packed storage

symmetric positive-definite matrix

LAPACK

2594
triangular matrix (continued)
symmetric positive-definite matrix (continued)
packed storage 584
ScalAPACK 1289
triangular distributed matrix 1291
triangular matrix
packed storage 601
Iterative Sparse Solvers 1832
Iterative Sparse Solvers based on Reverse Communication Interface (RCI ISS) 1832

J
Jacobi plane rotations 1019
Jacobian matrix calculation routines
?jacobi 2387
?jacobi_delete 2386
?jacobi_init 2384
?jacobi_solve 2385
?jacobix 2388

utility functions and routines (continued)
auxiliary routines (continued)
?lartgs 1141
?lascl 1141
?lasd0 1143
?lasd1 1144
?lasd2 1146
?lasd3 1149
?lasd4 1151
?lasd5 1152
?lasd6 1153
?lasd7 1156
?lasd8 1159
?lasd9 1160
?lasda 1162
?lasdq 1164
?lasdt 1166
?laset 1167
?lasrt 1168
?laswp 1169
?laum 1193
?orbbdb/?unbbdb 931
?orcstr?uncstr 1032
?orcstr2by1/?unctr2by1 1035
?sfkr 1197
?sum 1105
?syr 1103
?tfttp 1202
?tfttr 1203
?tpf 1210
?tptr 1211
?tptrf 1212
?tptrr 1213
i?max 1104
LAPACK_get_nancheck 1120
LAPACK_set_nancheck 1120
banded matrix equilibration
?gbequ 606
?gbequb 607
bidiagonal divide and conquer 1166
block reflector
triangular factor 1129
complex Hermitian matrix 1117
complex matrix multiplication 1000, 1215
complex symmetric matrix
symmetric rank-1 update 1103
complex vector
1-norm using true absolute value 1105
index of element with max absolute value 1104
linear transformation 1120
complex vector conjugation 1100
condition number estimation
?disna 833
?gbcon 501
?gcon 499
?gtcon 503
?hecon 513
?hecon 517
?hpccon 507
?hpcon 504
?hpccon 506
?ptcon 508
?spcon 516
?spcon 510
?spcon 510
?tbcon 521
?tpcon 520
?tpcon 520
diagonally dominant triangular factorization

2595
<table>
<thead>
<tr>
<th>Utility Functions and Routines (Continued)</th>
<th>Utility Functions and Routines (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagonally Dominant Triangular Factorization (Continued)</td>
<td>Symmetric Eigenproblems (Continued)</td>
</tr>
<tr>
<td>?dttrf 425</td>
<td>solving linear equations (Continued)</td>
</tr>
<tr>
<td>Driver Routines</td>
<td>?posvxx 660</td>
</tr>
<tr>
<td>Generalized LLS Problems</td>
<td>?ppsv 666</td>
</tr>
<tr>
<td>?ggglm 945</td>
<td>?ptsxv 668</td>
</tr>
<tr>
<td>?ggles 942</td>
<td>?ptsv 677</td>
</tr>
<tr>
<td>Generalized Nonsymmetric Eigenproblems</td>
<td>Generalized Symmetric Definite Eigenproblems</td>
</tr>
<tr>
<td>?gges 1076</td>
<td>?sysvxx 691</td>
</tr>
<tr>
<td>?ggesx 1080</td>
<td></td>
</tr>
<tr>
<td>?geev 1088</td>
<td></td>
</tr>
<tr>
<td>?ggevx 1091</td>
<td></td>
</tr>
<tr>
<td>Generalized Symmetric Definite Eigenproblems</td>
<td></td>
</tr>
<tr>
<td>?hgvd 1068</td>
<td>symmetric eigenproblems</td>
</tr>
<tr>
<td>?hbgvx 1073</td>
<td>?hbevd 979</td>
</tr>
<tr>
<td>?hgev 1039</td>
<td>?hbev 983</td>
</tr>
<tr>
<td>?hegv 1042</td>
<td>?heev 949</td>
</tr>
<tr>
<td>?hegvx 1047</td>
<td>?heevd 951</td>
</tr>
<tr>
<td>?hpgv 1052</td>
<td>?heevr 960</td>
</tr>
<tr>
<td>?hpgvd 1055</td>
<td>?hpev 965</td>
</tr>
<tr>
<td>?hpgvx 1060</td>
<td>?hpevd 968</td>
</tr>
<tr>
<td>?sbvg 1063</td>
<td>?hpevx 972</td>
</tr>
<tr>
<td>?sbvd 1066</td>
<td>?sbev 974</td>
</tr>
<tr>
<td>?sbvx 1070</td>
<td>?sbev 977</td>
</tr>
<tr>
<td>linear least squares problems</td>
<td>?sbev 980</td>
</tr>
<tr>
<td>?gelss 939</td>
<td>?spev 963</td>
</tr>
<tr>
<td>nonsymmetric eigenproblems</td>
<td>?spev 966</td>
</tr>
<tr>
<td>?geevx 994</td>
<td>?spev 969</td>
</tr>
<tr>
<td>?gesx 997</td>
<td>?stev 986</td>
</tr>
<tr>
<td>?geevx 1002</td>
<td>?stevd 987</td>
</tr>
<tr>
<td>?latms 1226</td>
<td>?stevr 991</td>
</tr>
<tr>
<td>General Matrix Equilibration</td>
<td>?stevx 989</td>
</tr>
<tr>
<td>?gejsv 1012</td>
<td>?syev 947</td>
</tr>
<tr>
<td>Singular Value Decomposition</td>
<td>?syev 950</td>
</tr>
<tr>
<td>?gsvd 1023</td>
<td>?syev 957</td>
</tr>
<tr>
<td>solving linear equations</td>
<td>general matrix</td>
</tr>
<tr>
<td>?dsb 652</td>
<td>lagge 1223</td>
</tr>
<tr>
<td>?gbvx 633</td>
<td>lagsy 1225</td>
</tr>
<tr>
<td>Generalized Eigenvalue Problems</td>
<td>?latms 1226</td>
</tr>
<tr>
<td>?gbvx 635</td>
<td></td>
</tr>
<tr>
<td>?gdgdv 640</td>
<td></td>
</tr>
<tr>
<td>?gsgd 619</td>
<td></td>
</tr>
<tr>
<td>?gsvd 621</td>
<td></td>
</tr>
<tr>
<td>?gsvd 626</td>
<td></td>
</tr>
<tr>
<td>?gst 647</td>
<td></td>
</tr>
<tr>
<td>?gsv 649</td>
<td></td>
</tr>
<tr>
<td>?hesv 698</td>
<td></td>
</tr>
<tr>
<td>Hermitian Indefinite Matrix Equilibration</td>
<td>generalized SVD</td>
</tr>
<tr>
<td>?hesv 703</td>
<td>?gsgv 913</td>
</tr>
<tr>
<td>?hesv 706</td>
<td>?tgsja 923</td>
</tr>
<tr>
<td>?hsyvd 717</td>
<td></td>
</tr>
<tr>
<td>?hpsvx 718</td>
<td></td>
</tr>
<tr>
<td>?pdsy 653</td>
<td></td>
</tr>
<tr>
<td>?pdsyrk 672</td>
<td></td>
</tr>
<tr>
<td>?pdsy 673</td>
<td></td>
</tr>
<tr>
<td>?post 653</td>
<td></td>
</tr>
<tr>
<td>?post 656</td>
<td></td>
</tr>
<tr>
<td>Hermitian Matrix</td>
<td></td>
</tr>
</tbody>
</table>

2596
utility functions and routines (continued)
  Hermitian matrix (continued)
    ?laghe 1224
    Hermitian positive-definite matrix equilibration
    ?poequ 609
    ?poequb 611
  Householder matrix
    elementary reflector 1127
  LQ factorization
    ?gelqf 609
    ?ormlq 606
    ?unmlq 647
    matrix equilibration
    ?pbequ 613
    ?ppequ 612
  matrix inversion
    ?getri 579
    ?hetri 586
    ?hetri2 589
    ?hetri2x 591
    ?hptri 592
    ?spetri 596
    ?sptri 598
    ?syetri 588
    ?syetri2 589
    ?syetri2x 590
    ?tpetri 601
  ?ttri 598
  mkl_?getrinp 580
  merging sets of singular values 1146, 1156
  mixed precision iterative refinement subroutines 619, 653
  mkl_?gepack_compact 269
  mkl_?getrfnp_compact 263
  mkl_?getrinp_compact 266
  mkl_?geunpack_compact 271
  mkl_?gmrqf 262, 265
  nonsymmetric eigenvalue problems
    ?gebak 858
    ?gebal 856
    ?gehrd 848
    ?hsein 862
    ?hseqr 859
    ?orghr 849
    ?ormr 851
    ?trevc 866
    ?trex 877
    ?ttsen 879
    ?trsna 873
    ?unghr 853
    ?unmhr 854
  off-diagonal and diagonal elements 1167
  permutation of matrix columns 1122
  permutation of matrix rows 1120
  plane rotation 1140, 1141
  QL factorization
    ?gell 478
    ?org2l 500
    ?orm2l 502
    ?unmlq 573
  QR factorization
    ?gemqrt 729
    ?geq2 733
  QR factorization (continued)
    QR factorization (continued)
      ?geq2 733
      ?geq2r 1106
      ?geqrf 724
      ?geqrfp 726
      ?geqrt 727
      ?geqrt2 1108
      ?geqrt3 1109
      ?ggqrf 767
      ?ggqrf 770
      ?orgqr 735
      ?ormqr 736
      ?tpmqrt 774
      ?tpqrt 772
      ?tpqrt2 1205
      ?tpqrt3 1207
      ?ungrq 738
      ?unmqr 739
      p?geqrf 1297
    random numbers vector 1134
    real lower bidiagonal matrix
    SVD 1164
    real symmetric matrix 1116
    real upper bidiagonal matrix
    singular values 1143
    SVD 1144, 1162, 1164
    reciprocal condition numbers for eigenvalues and/or eigenvectors
      ?tgsna 909
    rectangular full packed format 431, 468
    RQ factorization
      ?gerqf 755
      ?orgqr 757
      ?ormqr 759
      ?unqr 758
      ?unmr 760
      RZ factorization
      ?ormr 764
      ?tzrzf 762
      ?unmr 765
      singular value decomposition
        ?bdsdc 793
        ?bdsqr 791
        ?gbbrd 780
        ?gebrd 778
        ?orgbr 783
        ?ormbr 784
        ?ungbr 787
        ?unmbr 788
      solution refinement and error estimation
        ?gbtrfs 531
        ?gbtrfs 533
        ?gerfs 523
        ?gerfs 525
        ?orgfs 529
        ?herfs 562
        ?hesdxa 564
        ?hprrsf 571
        ?hprrsf 573
        ?pprsf 549
        ?pprsf 552
        ?prsfs 569
        ?syrsf 554
        ?sysrfsx 556
        ?tprsfs 576
utility functions and routines (continued)
solution refinement and error estimation (continued)
?tprs 574
?trevc3 869
?trrfs 572
solving linear equations
?dtrsb 465
?gbtrs 461
?getrs 459
?gttrs 463
?heswapr 1196
?hetrs 480
?hetrs_rook 483
?hesr 2 485
?hptrs 491
?pbrtrs 472
?pptrs 468
?potrs 467
?pttrs 470
?pttrs 474
?pptrs 490
?syswapr 1194
?sytrs 475
?sytrs_rook 478
?sytrs 484
?tbtrs 497
?tptrs 495
?trtrs 493
sorting numbers 1168
square root 1123
square roots 1149, 1151, 1152, 1159, 1160
Syldesyltien equation
?trsyl 882
symmetric eigenvalue problems
?dsna 833
?hbtrd 815
?hetrd 801
?hpbrd 809
?opgr 806
?opmtr 807
?orgtr 798
?ormtr 799
?poeq 827
?tpbrd 813
?spbrd 805
?stebr 829
?stecq 823
?steqr 824
?stein 831
?stemr 820
?steqr 817
?sterf 816
?ungtr 802
?unmtr 803
?upgq 810
?upmtr 811
symmetric indefinite matrix equilibration
?sysyed 865
symmetric positive-definite matrix equilibration
?poeq 609
?poeq 611
test routines
?lagge 1223
?laghe 1224
?lagh 825
?latms 1226
trapezoidal matrix 1118
tridiagonal symmetric positive-definite matrix (continued)
band matrix (continued)
LAPACK 633, 635
ScalAPACK 1435
banded matrix
extra precise iterative refinement
LAPACK 640
extra precise iterative refinement 533
LAPACK 640
Cholesky-factored matrix
LAPACK 472
diagonally dominant tridiagonal matrix
LAPACK 465, 652
diagonally dominant-like matrix
banded 1257
tridiagonal 1260
general band matrix
ScalAPACK 1437
general matrix
band storage 461, 1255
extra precise iterative refinement 525
general tridiagonal matrix
ScalAPACK 1440
Hermitian indefinite matrix
extra precise iterative refinement
LAPACK 706
LAPACK 706
Hermitian matrix
error bounds 703, 718
packed storage 491, 717, 718
Hermitian positive-definite matrix
band storage
LAPACK 672
ScalAPACK 1449
error bounds
LAPACK 656
ScalAPACK 1444
extra precise iterative refinement
LAPACK 660
LAPACK linear equations, solving
multiple right-hand sides
symmetric positive-definite matrix 653
packed storage 470, 666, 668
ScalAPACK 1444
Hermitian positive-definite tridiagonal linear equations 1729
Hermitian positive-definite tridiagonal matrix 1265
multiple right-hand sides
band matrix
LAPACK 633, 635
ScalAPACK 1435
banded matrix
LAPACK 640
diagonally dominant tridiagonal matrix 652
Hermitian indefinite matrix
LAPACK 706
Hermitian matrix 698, 717
Hermitian positive-definite matrix
band storage 672
square matrix
LAPACK 619, 621, 626
ScalAPACK 1428, 1430
symmetric indefinite matrix
LAPACK 691
symmetric matrix 681, 684, 712
tridiagonal symmetric positive-definite matrix (continued)
LAPACK 660
tridiagonal matrix 647, 649
overestimated or underestimated system 1453
tridiagonal symmetric positive-definite matrix (continued)
symmetric/Hermitian positive-definite matrix (continued)
symmetric positive-definite matrix
band storage 672
symmetric/Hermitian positive-definite matrix
LAPACK 660
tridiagonal matrix 647, 649
ScaLAPACK auxiliary 1728
tridiagonal symmetric positive-definite matrix
error bounds 678
LAPACK 677
ScalAPACK 1451
triangular matrix
band storage 497, 1655
packed storage 495
tridiagonal Hermitian positive-definite matrix
error bounds 678
LAPACK 677
ScalAPACK 1451
triangular matrix
error bounds 649
LAPACK 463, 474, 647, 649
ScalAPACK auxiliary 1728
triangular symmetric positive-definite matrix
error bounds 678
LAPACK 677
ScalAPACK 1451
Linear Least Squares (LLS) Problems 935
LoadStreamF 2048
LoadStreamM 2050
Lognormal 2078
LQ factorization
computing the elements of orthogonal matrix Q 742
real orthogonal matrix Q 1312
unitary matrix Q 745, 1314
general rectangular matrix 1105, 1531
Isame 2409
Isamen 2409
LU factorization
band matrix
blocked algorithm 1725
unblocked algorithm 1724
Index 2599
solving linear equations (continued)
diagonally dominant tridiagonal matrix 425
diagonally dominant-like tridiagonal matrix 1245
general matrix 1111, 1539
solving linear equations
square matrix 1430
triangular band matrix 1519
tridiagonal band matrix 1521
tridiagonal matrix 424, 1727
with partial pivoting 1111, 1539

M
machine parameters
LAPACK 1221
ScaLAPACK 1743
matrices, copying
general matrix 1746
trapezoidal matrix 1748
matrix arguments
column-major ordering 2539, 2542
example 2543
leading dimension 2542
number of columns 2542
number of rows 2542
transposition parameter 2542
matrix converters
mkl_?csrsbr 227
mkl_?csrcoo 225
mkl_?csrcsc 229
mkl_?csrdia 231
mkl_?csrsy 233
mkl_?ldnscsr 222
matrix equation
AX = B 121, 363, 411, 459, 1200, 1237, 1253
matrix one-dimensional substructures 2539
matrix-matrix operation
product
general distributed matrix 102, 353, 356, 360
rank-2 update
Hermitian distributed matrix 2299
Hermitian matrix 109, 257, 260, 262
symmetric distributed matrix 2306
symmetric matrix 116
rank-k update
Hermitian matrix 107
symmetric distributed matrix 2304
rank-n update
symmetric matrix 114
scalar-matrix-matrix product
Hermitian distributed matrix 2295
Hermitian matrix 105
symmetric distributed matrix 2301
symmetric matrix 112
matrix-matrix operation: scalar-matrix-matrix product
triangular distributed matrix 2312
matrix-matrix operation: scalar-matrix-matrix product
triangular matrix 119
matrix-vector operation
product
Hermitian matrix 67, 70, 74
real symmetric matrix 81, 85
triangular matrix 89, 95, 98
rank-1 update
Hermitian matrix 71, 75
real symmetric matrix 82, 87
rank-2 update
rank-2 update (continued)
Hermitian matrix 72, 77
symmetric matrix 84, 88
matrix-vector operation: product
Hermitian matrix
band storage 67
packed storage 74
real symmetric matrix
packed storage 81
symmetric matrix
band storage 78
triangular matrix
band storage 89
packed storage 95
matrix-vector operation: rank-1 update
Hermitian matrix
packed storage 75
real symmetric matrix
packed storage 82
matrix-vector operation: rank-2 update
Hermitian matrix
packed storage 77
symmetric matrix
packed storage 84
mkl_?bsrgemv 142
mkl_?bsrsmm 191
mkl_?bsrmv 175
mkl_?bssrm 205
mkl_?bssrv 183
mkl_?bssymv 148
mkl_?bsrsv 154
mkl_?coogemv 144
mkl_?coomm 196
mkl_?coomv 179
mkl_?coosm 203
mkl_?coosv 187
mkl_?coosymv 150
mkl_?cootrsv 156
mkl_?cootrsv 156
mkl_?cscmm 194
mkl_?cscmv 177
mkl_?cscsm 201
mkl_?cscsv 185
mkl_?csradd 235
mkl_?csrsbr 227
mkl_?csrcoo 225
mkl_?csrcsc 229
mkl_?csrdia 231
mkl_?csrgemv 141
mkl_?csrsmm 189
mkl_?csrsmultcsr 238
mkl_?csrsmultd 241
mkl_?csrsmv 173
mkl_?csrsky 233
mkl_?csrsm 199
mkl_?csrsv 181
mkl_?csrsymv 147
mkl_?csrtrsv 152
mkl_?diagemv 145
mkl_?diamm 214
mkl_?diamv 207
mkl_?diasm 219
mkl_?diav 211
mkl_?diasymv 151
mkl_?diatrsv 158
mkl_?dnscsr 222
mkl_?gemm_compact 257, 260
mkl_?gepack_compact 269

2600
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkl_sparse_s_export_bsr</td>
<td>291</td>
</tr>
<tr>
<td>mkl_sparse_s_export_csc</td>
<td>289</td>
</tr>
<tr>
<td>mkl_sparse_s_export_csr</td>
<td>288</td>
</tr>
<tr>
<td>mkl_sparse_s_mm</td>
<td>319</td>
</tr>
<tr>
<td>mkl_sparse_s_spmm</td>
<td>293</td>
</tr>
<tr>
<td>mkl_sparse_s_spmmd</td>
<td>328</td>
</tr>
<tr>
<td>mkl_sparse_s_trsm</td>
<td>322</td>
</tr>
<tr>
<td>mkl_sparse_s_trsv</td>
<td>317</td>
</tr>
<tr>
<td>mkl_sparse_s_update_values</td>
<td>293</td>
</tr>
<tr>
<td>mkl_sparse_set_lu_smoother_hint</td>
<td>296</td>
</tr>
<tr>
<td>mkl_sparse_set_memory_hint</td>
<td>309</td>
</tr>
<tr>
<td>mkl_sparse_set_mm_hint</td>
<td>302</td>
</tr>
<tr>
<td>mkl_sparse_set_mv_hint</td>
<td>298</td>
</tr>
<tr>
<td>mkl_sparse_set_sm_hint</td>
<td>304</td>
</tr>
<tr>
<td>mkl_sparse_set_sv_hint</td>
<td>300</td>
</tr>
<tr>
<td>mkl_sparse_spmm</td>
<td>327</td>
</tr>
<tr>
<td>mkl_sparse_sypr</td>
<td>333</td>
</tr>
<tr>
<td>mkl_sparse_z_add</td>
<td>325</td>
</tr>
<tr>
<td>mkl_sparse_z_create_bsr</td>
<td>282</td>
</tr>
<tr>
<td>mkl_sparse_z_create_coo</td>
<td>280</td>
</tr>
<tr>
<td>mkl_sparse_z_create_csc</td>
<td>278</td>
</tr>
<tr>
<td>mkl_sparse_z_create_csr</td>
<td>277</td>
</tr>
<tr>
<td>mkl_sparse_z_export_bsr</td>
<td>291</td>
</tr>
<tr>
<td>mkl_sparse_z_export_csc</td>
<td>289</td>
</tr>
<tr>
<td>mkl_sparse_z_export_csr</td>
<td>288</td>
</tr>
<tr>
<td>mkl_sparse_z_mm</td>
<td>319</td>
</tr>
<tr>
<td>mkl_sparse_z_spmm</td>
<td>327</td>
</tr>
<tr>
<td>mkl_sparse_z_set_value</td>
<td>293</td>
</tr>
<tr>
<td>mkl_sparse_z_spmmd</td>
<td>328</td>
</tr>
<tr>
<td>mkl_sparse_z_trsm</td>
<td>322</td>
</tr>
<tr>
<td>mkl_sparse_z_trsv</td>
<td>317</td>
</tr>
<tr>
<td>mkl_sparse_z_update_values</td>
<td>293</td>
</tr>
<tr>
<td>mkl_sparse_spmm</td>
<td>327</td>
</tr>
<tr>
<td>mkl_sparse_sypr</td>
<td>333</td>
</tr>
<tr>
<td>mkl_sparse_z_add</td>
<td>325</td>
</tr>
<tr>
<td>mkl_sparse_z_create_bsr</td>
<td>282</td>
</tr>
<tr>
<td>mkl_sparse_z_create_coo</td>
<td>280</td>
</tr>
<tr>
<td>mkl_sparse_z_create_csc</td>
<td>278</td>
</tr>
<tr>
<td>mkl_sparse_z_create_csr</td>
<td>277</td>
</tr>
<tr>
<td>mkl_sparse_z_export_bsr</td>
<td>291</td>
</tr>
<tr>
<td>mkl_sparse_z_export_csc</td>
<td>289</td>
</tr>
<tr>
<td>mkl_sparse_z_export_csr</td>
<td>288</td>
</tr>
<tr>
<td>mkl_sparse_z_mm</td>
<td>319</td>
</tr>
<tr>
<td>mkl_sparse_z_spmm</td>
<td>327</td>
</tr>
<tr>
<td>mkl_sparse_z_set_value</td>
<td>293</td>
</tr>
<tr>
<td>mkl_sparse_z_spmmd</td>
<td>328</td>
</tr>
<tr>
<td>mkl_sparse_z_trsm</td>
<td>322</td>
</tr>
<tr>
<td>mkl_sparse_z_trsv</td>
<td>317</td>
</tr>
<tr>
<td>mkl_sparse_z_update_values</td>
<td>293</td>
</tr>
<tr>
<td>MKL_Thread_Free_Buffers</td>
<td>2414</td>
</tr>
<tr>
<td>mkl_tppack</td>
<td>1216</td>
</tr>
<tr>
<td>mkl_tpunpack</td>
<td>1218</td>
</tr>
<tr>
<td>mkl_verbosse</td>
<td>2436</td>
</tr>
<tr>
<td>mkl_verbosse_output_file</td>
<td>2437</td>
</tr>
<tr>
<td>mkl_zgemm_compact</td>
<td>258</td>
</tr>
<tr>
<td>MKLFreeTls 2011</td>
<td></td>
</tr>
<tr>
<td>MPI</td>
<td>1234</td>
</tr>
<tr>
<td>Multinomial 2104</td>
<td></td>
</tr>
<tr>
<td>Multiplicative Congruential Generator 2027</td>
<td></td>
</tr>
</tbody>
</table>

**N**

naming conventions

BLAS 41
LAPACK 1237
Nonlinear Optimization Solvers 2369
PBLAS 2244
Sparse BLAS Level 1 123
Sparse BLAS Level 2 131
Sparse BLAS Level 3 131
VM 1895

negative eigenvalues 1555
NegBinomial 2103
NewStream 2038
NewStreamEx 2040
NewTaskX1D 2121
Nonsymmetric Eigenproblems 994
numroc 1745

**O**

off-diagonal elements

off-diagonal elements (continued)

initialization 1614
LAPACK 1167
ScaLAPACK 1614

orthogonal matrix

CS decomposition
LAPACK 928, 931, 1032, 1035
from LQ factorization
ScaLAPACK 1638
from QR factorization
ScaLAPACK 1634, 1642
from RQ factorization
ScaLAPACK 1636

**P**

p?agemv 2261
p?ahemv 2270
p?amax 2246
p?asum 2247
p?asumv 2277
p?atrmv 2285
p?axpy 2248
p?copy 2250
p?dsbv 1437
p?dbsv 1437
p?dbrt 1243
p?dbrts 1257
p?dt and 1519
p?dot 2251
p?dotc 2252
p?dotu 2253
p?dtzv 1440
p?dttri 1245
p?dttrs 1260
p?dtzsv 1521
p?gbsv 1435
p?gbtrf 1241
p?gbtrs 1255
p?geadd 2290
p?gebd2 1526
p?gebld 1413
p?gecon 1270
p?geeq 1292
p?gehdo 1529
p?gehdd 1398
p?gelq 1531
p?gelm 1310
p?gels 1453
p?gemm 2293
p?gemr2d 1746
p?genv 2259
p?geq 1533
p?geqlf 1321
p?geqpf 1299
p?geqr2 1535
p?geqrf 1297
p?ger 2263
p?gerc 2265
p?gerf 1277
p?gerq 1537
p?geqf 1332
p?geru 2267
p?gesv 1428
p?gesvd 1489
p?gesvx 1430
p?getrf 1539
Packed formats 2204
packed storage scheme 2541
parallel direct solver (PARDISO) 1752
Parallel Direct Sparse Solver for Clusters 1794
parallel direct sparse solver interface
  mkl_pardiso_pivot 1768
  pardiso 1759
  pardiso_64 1766
  pardiso_getdiag 1769
  pardiso_getenv 1767
  pardiso_handle_delete 1774
  pardiso_handle_delete_64 1776
  pardiso_handle_restore 1773
  pardiso_handle_restore_64 1775
  pardiso_handle_store 1772
  pardiso_handle_store_64 1774
  pardiso_setenv 1767
  pardisoinit 1765
parallel direct sparse solver interface for clusters
  cluster_sparse_solver 1796
  cluster_sparse_solver_64 1801
parameters
  for a Givens rotation 52
  modified Givens transformation 53
  pardiso 1759
  pardiso iparm parameter 1781
  PARnSO parameters 1777
  pardiso_64 1766
  PARDISO_DATA_TYPE 1794
  pardiso_export 1770
  pardiso_getdiag 1769
  pardiso_getenv 1767
  pardiso_handle_delete 1774
  pardiso_handle_delete_64 1776
  pardiso_handle_restore 1773
  pardiso_handle_restore_64 1775
  pardiso_handle_store 1772
  pardiso_handle_store_64 1774
  pardisoinit 1765
Partial Differential Equations support
  Helmholtz problem on a sphere 2335
  Poisson problem on a sphere 2335
  three-dimensional Helmholtz problem 2337
  three-dimensional Laplace problem 2338
  three-dimensional Poisson problem 2338
  two-dimensional Helmholtz problem 2334
  two-dimensional Laplace problem 2335
  two-dimensional Poisson problem 2335
PBLAS Level 1 functions
  p?amax 2246
  p?asum 2247
  p?dot 2251
  p?dotc 2252
  p?dotu 2253
  p?nrm2 2254
PBLAS Level 1 routines
  p?amax 2246
  p?asum 2246
PBLAS Level 1 routines (continued)
  p?axpy 2246, 2248
  p?copy 2246, 2250
  p?dot 2246
  p?dotc 2246
  p?dotu 2246
  p?nrm2 2246
  p?scal 2246, 2255
  p?swap 2246, 2256
PBLAS Level 2 routines
  ?agemv 2258
  ?asymv 2258
  ?gemv 2258
  ?ger 2258
  ?gerc 2258
  ?geru 2258
  ?hemv 2258
  ?her 2258
  ?her2 2258
  ?symv 2258
  ?syrmv 2258
  ?syrv 2258
  ?trmv 2258
  ?tsrv 2258
  ?agemv 2261
  ?ahemv 2270
  ?asymv 2277
  ?atrmv 2285
  ?gemv 2259
  ?ger 2263
  ?gerc 2265
  ?geru 2267
  ?hemv 2268
  ?her 2272
  ?her2 2274
  ?symv 2276
  ?syrv 2279
  ?syrv2 2281
  ?trmv 2283
  ?tsrv 2287
PBLAS Level 3 routines
  ?geadd 2290
  ?gemm 2289, 2293
  ?hemm 2289, 2295
  ?her2k 2289, 2299
  ?herk 2289, 2297
  ?symm 2289, 2301
  ?symm2 2289, 2306
  ?syrk 2289, 2304
  ?tradd 2291
  ?tran 2308
  ?tranu 2309
  ?trmm 2289, 2312
  ?trsm 2289, 2314
PBLAS routines
  routine groups 2243
  pccgemv 2261
  pcahemv 2270
  pcamax 2246
  pcatrmv 2285
  pccopy 2250
  pcdotc 2252
  pcdotu 2253
  pcdotu 2253
  pceadd 2290
  pcecon 1270
  pcgeam 2293
Index

pcgemv 2259
pcgerc 2265
pcgeru 2267
pcheevr 1477
pchemm 2295
pchemv 2268
pchengst 1377
pchentrd 1379
pcher 2272
pcher2 2274
pcher2k 2299
pcherk 2297
pclapv2 1567
pclase2 1613
pcnrm2 2254
pcscal 2255
pcsscal 2255
pcswap 2256
pcsymm 2301
pcsyr2k 2306
pcsyrk 2304
pctradd 2291
pctranu 2309
pctrevc 1410
pctrmm 2312
pctrmv 2283
pctrsm 2314
pctrsv 2287
pcunmr3 1341
pdagemv 2261
pdamax 2246
pdasum 2247
pdasymv 2277
pdatrmv 2285
pdaxpy 2248
pdcopy 2250
pddot 2251
PDE support 2317
pdgeadd 2290
pdgebal 1524
pdgecon 1270
pdgemm 2293
pdgemv 2259
pdger 2263
pdlaiectb 1555
pdlaiectl 1555
pdlamve 1556
pdlapv2 1567
pdlaqr0 1570
pdlaqr1 1573
pdlaqr2 1576
pdlaqr3 1579
pdlaqr5 1582
pdlase2 1613
pdlasrt 1617
pdnrm2 2254
pdormr3 1338
pdrot 1663
pdscal 2255
pdstedc 1392
pdswap 2256
pdsyevr 1462
pdsymm 2301
pdsymv 2276
pdsyngst 1365
pdsyntrd 1367
pdsyr 2279
pdsyr2 2281

pdsyr2k 2306
pdsyrk 2304
pdtradd 2291
pdtran 2308
pdtranc 2310
pdtrmm 2312
pdtrmv 2283
pdtrord 1672
pdtrsen 1676
pdtrsm 2314
pdtrsv 2287
pdzasum 2247
permutation matrix 2522
picopy 2250
pilaenvx 1735
pilaver 1514
pivoting matrix rows or columns 1565
pjlaenv 1737
planar rotation 1663
pmpcol 1515
pmpim2 1516
points rotation
in the modified plane 52
in the plane 51
Poisson 2099
Poisson problem
on a sphere 2335
three-dimensional 2338
two-dimensional 2335
Poisson Solver
routines
?_commit_Helmholtz_2D 2344
?_commit_Helmholtz_3D 2344
?_commit_sph_np 2353
?_commit_sph_p 2353
?_Helmholtz_2D 2347
?_Helmholtz_3D 2347
?_init_Helmholtz_2D 2341
?_init_Helmholtz_3D 2341
?_init_sph_np 2351
?_init_sph_p 2351
?_sph_np 2355
?_sph_p 2355
free_Helmholtz_2D 2350
free_Helmholtz_3D 2350
free_sph_np 2357
free_sph_p 2357
structure 2333
Poisson Solver Interface 2332
PoissonV 2101
pprfs 549
pptrs 470
preconditioned Jacobi SVD 1012
preconditioners based on incomplete LU factorization
dcsrilu0 1859
dcsrilut 1862
Preconditioners Interface Description 1858
process grid 1234, 1235, 2243
product
distributed matrix-vector
general matrix 2259, 2261
distributed vector-scalar 2255
matrix-vector
distributed Hermitian matrix 2268, 2270
distributed symmetric matrix 2276, 2277
distributed triangular matrix 2283, 2285
general matrix 59, 61
Hermitian matrix 67, 70, 74

2605


**scalar-matrix-matrix (continued)**
- matrix-vector (continued)
- real symmetric matrix 81, 85
- triangular matrix 89, 95, 98
- scalar-matrix
- general distributed matrix 2293
- general matrix 102, 353, 356, 360
- Hermitian distributed matrix 2295
- Hermitian matrix 105
- scalar-matrix-matrix
- general distributed matrix 2293
- general matrix 102, 353, 356, 360
- Hermitian distributed matrix 2295
- Hermitian matrix 105
- triangular matrix 89, 95, 98
- Hermitian distributed matrix
- Hermitian matrix
- symmetric distributed matrix
- symmetric matrix
- triangular distributed matrix
- triangular matrix
- vector-scalar
- product:matrix-vector
- general matrix
- band storage
- Hermitian matrix
- packed storage
- Hermitian matrix
- packed storage
- symmetric matrix
- triangular matrix
- packed storage
- triangular matrix

**Hermitian distributed matrix**
- **Hermitian matrix**

**vector-scalar**

**product:matrix-vector**
- general matrix
- band storage
- packed storage
- real symmetric matrix
- packed storage
- symmetric matrix
- triangular matrix
- packed storage

**Hermitian matrix**
- band storage
- packed storage

**real symmetric matrix**
- band storage
- packed storage

**scalar-matrix**
- **general distributed matrix**
- **general matrix**
- **Hermitian distributed matrix**
- **Hermitian matrix**
- **symmetric distributed matrix**
- **symmetric matrix**
- **triangular distributed matrix**
- **triangular matrix**

**block storage**

**pseudorandom numbers**

**psagemv** 2261
- psamax 2246
- psasum 2247
- psasymv 2276
- psaxpy 2248
- pscasum 2247
- psdot 2251
- pseudorandom numbers 2025
- psgeadd 2290
- psgebal 1524
- psgecon 1270
- psgeev 2259
- psger 2265
- pslaiect 1555
- psllmv 1556
- pslapv2 1567
- pslasr 1579
- psllq 1582
- psllx 1582
- psrmm 2254
- psormr 1338
- pssrot 1663
- psscal 2255
- psstdec 1392
- psstl 2256
- pssyev 1462
- pssymm 2301
- pssyev 2276
- pssyngst 1365
- pssynrtrd 1367
- pssyr 2279
- pssyev 2281
- pssyrk 2230
- pssyev 2304
- pstradd 2291
- pstran 2308
- pstranc 2310
- pstrmm 2312
- pstrmv 2283
- pstrord 1672
- pstrsen 1676
- pstrsm 2314
- pstrsv 2287
- pxerbla 2406
- pzhemm 2261
- pzhemv 2270
- pzamax 2246
- pztmmv 2285
- pzxaxpy 2248
- pzaxpy 2248
- pzdcopy 2250
- pzdcopy 2253
- pzdscal 2255
- pzeadd 2290
- pzecon 1270
- pzemgm 2293
- pzemgm 2259
- pzhcer 2265
- pzhger 2267
- pzhheev 1477
- pzhemm 2295
- pzhemv 2268
- pzhengst 1377
- pzhentr 1379
- pzhger 2277
- pzhger 2274
- pzhger 2299
- pzhgk 2297
- pzlavg 1567
- pzlase 1613
- pznrmr 2254
- pzsca 2255
- pzsca 2256
- pzsca 2301
- pzsyrk 2306
- pzsyr 2304
- pztadd 2291
- pztanu 2309
- pztrevc 1410
- pztmm 2312
- pztmm 2283
- pztmm 2314
- pztmmv 2287
- pztmmv 2314

**Q**

**QL factorization**
- computing the elements of complex matrix Q 751
- orthogonal matrix Q 1323
- real matrix Q 750
- unitary matrix Q 1325
- general rectangular matrix
- ScalAPACK 1533
- multiplying general matrix by orthogonal matrix Q 1327
multiplying general matrix by \((continued)\) 
multiplying general matrix by \((continued)\)
unitary matrix \(Q\)

QR factorization

applying matrix obtained from block reflector to

general matrix

orthogonal/unitary matrix \(Q\)

computing the elements of

orthogonal matrix \(Q\)

general rectangular matrix

LAPACK 1106

ScaLAPACK 1535, 1537

multiplying general matrix by

orthogonal/unitary matrix \(Q\)

with pivoting

ScaLAPACK 1299

quasi-random numbers 2025

quasi-triangular matrix

LAPACK 846, 884

ScaLAPACK 1398

R

random number generators 2025

random stream 2034

random stream descriptor 2026

Random Streams 2034

rank-1 update

conjugated, distributed general matrix 2265

conjugated, general matrix 64

distributed general matrix 2263

distributed Hermitian matrix 2272

distributed symmetric matrix 2279

general matrix 63

Hermitian matrix

packed storage 75

real symmetric matrix

packed storage 82

unconjugated, distributed general matrix 2267

unconjugated, general matrix 66

rank-2 update

distributed Hermitian matrix 2274

distributed symmetric matrix 2281

Hermitian matrix

packed storage 77

symmetric matrix

packed storage 84

rank-2k update

Hermitian distributed matrix 2299

Hermitian matrix 109, 257, 260

symmetric distributed matrix 2306

symmetric matrix 116

rank-k update

distributed Hermitian matrix 2297

Hermitian matrix 107

symmetric distributed matrix 2304

rank-n update

symmetric matrix 114

Rayleigh 2076

RCI CG Interface 1833

RCI CG sparse solver routines

dcg 1846, 1850

dcg_get 1847

dcgmrhs_get 1852

RCI CG sparse solver routines:dcg_check 1845

RCI CG sparse solver routines:dcg_init 1844

RCI CG sparse solver routines:dcgmrhs_check 1849

RCI CG sparse solver routines:dcgmrhs_init 1848

RCI FGMRES Interface 1838

RCI FGMRES sparse solver routines
dfgmres_get 1856

RCI FGMRES sparse solver routines:dfgmres_check 1853

RCI FGMRES sparse solver routines:dfgmres_init 1852

RCI GFMRES sparse solver routines
dfgres 1854

RCI ISS 1832

RCI ISS interface 1832

RCI ISS sparse solver routines

implementation details 1857

real symmetric matrix

1-norm value 1116

infinity- norm 1116

largest absolute value of element 1116

reducing generalized eigenvalue problems

LAPACK 835

ScaLAPACK 1424

reduction to upper Hessenberg form

general matrix 1529

refining solutions of linear equations

band matrix 531

banded matrix 533

general matrix 523, 525, 1277

Hermitian indefinite matrix 564

Hermitian matrix

packed storage 571

Hermitian positive-definite matrix

band storage 550

packed storage 549

symmetric indefinite matrix 556

symmetric matrix

packed storage 569

symmetric positive-definite matrix

band storage 550

packed storage 549

symmetric/Hermitian positive-definite distributed

matrix 1281

tridiagonal matrix 539

RegisterBrng 2107

registering a basic generator 2106

Relatively robust representation (RRR) 1704

reordering of matrices 2523

Reverse Communication Interface 1832

rotation

of points in the modified plane 52

of points in the plane 51

of sparse vectors 129

parameters for a Givens rotation 52

parameters of modified Givens transformation 53

rotation, planar 1663

routine name conventions

BLAS 41

Nonlinear Optimization Solvers 2369

PBLAS 2244

Sparse BLAS Level 1 123

Sparse BLAS Level 2 131

Sparse BLAS Level 3 131

RQ factorization

computing the elements of

complex matrix \(Q\)

orthogonal matrix \(Q\)

real matrix \(Q\)

unitary matrix \(Q\)
utility functions and routines (continued)

auxiliary routines (continued)

p?larzt 1609
p?lascl 1611
p?lascl 1614
p?lasmsub 1616
p?lassq 1618
p?laswp 1620
p?latra 1622
p?latrd 1623
p?latrs 1626
p?latrz 1628
p?lauu2 1630
p?lauum 1631
p?lawil 1633
p?max1 1513
p?org2l/p?ung2l 1634
p?org2r/p?ung2r 1636
p?org2l/p?ung2l 1638
p?org2r/p?ung2r 1640
p?orm2l/p?unm2l 1642
p?orm2r/p?unm2r 1645
p?orm2l/p?unm2l 1649
p?orm2r/p?unm2r 1652
p?pbtrsv 1655
p?potf2 1661
p?pttrsv 1659
p?rsc1 1665
p?sum1 1518
p?sygs2/p?hegs2 1666
p?sytd2/p?hetd2 1669
p?trti2 1680
pdgebal 1524
dlaiectb 1555
ndlaiect 1555
ndlamev 1556
pdlaq0 1570
pdlaq1 1573
pdlaq2 1576
pdlaq3 1579
pdlaq5 1582
pdrot 1663
pdtrd 1672
pdtrees 1676
pmpcol 1515
pmpsim 1516
psgebal 1524
pslaiect 1555
pslamev 1556
pslaqr0 1570
pslaqr1 1573
pslaqr2 1576
pslaqr3 1579
pslaqr5 1582
psrot 1663
pstree 1672
pstrsen 1676
slaqr6 1685
slar1va 1688
slarb2 1692
slarbd 1694
slarfe2a 1701
slarfe2b 1719

ScaLAPACK routines

1D array redistribution 1586, 1587
auxiliary routines
?combamax1 1517
?dbrf2 1724
?dbtf2 1725
?ddtf 1727
?ddtrsv 1728
?lamsh 1683
?lasrte 1710
?lasrt2 1711
?pattrsv 1729
?stein2 1722
?steqr2 1731
dlaqr6 1685
dlar1va 1688
dlarb2 1692
dlarrd 1694
dlarre 1697
dlarre2a 1701
dlarrf 1704
dlarrv 1706
dstegr2 1712
dstegr2a 1715
dstegr2b 1719
?p?dbtrsv 1519
?p?ddtrsv 1521
?p?gebd2 1526
?p?gehd2 1529
?p?geq2 1531
?p?geql2 1533
?p?geqr2 1535
?p?geq2 1537
?p?getf2 1539
?p?latrd 1541
?p?lacv 1513
?p?laco 1544
?p?lacosb 1545
?p?lacer 1547
?p?lacp2 1548
?p?laca 1550
?p?laev 1551
?p?laahd 1553
?p?lahe 1555
?p?lange 1557
?p?lanhs 1559
?p?lanl1, p?lanhe 1561
?p?lantr 1563
?p?lapiv 1565
?p?lajoq 1569
?p?laseq 1584
?p?laev 1586
?p?laev 1587
?p?larf 1588
?p?larb 1590
?p?larc 1593
?p?larf 1596
?p?lart 1598
?p?lar 1600
?p?larz 1603
?p?larz 1606

ScaLAPACK
sbbcsd 928
sbdsdc 793
SaveStreamF 2047
SaveStreamM 2049
utility functions and routines (continued)
utility functions and routines (continued)

block reflector
triangular factor 1598, 1609
Cholesky factorization 1251
complex matrix
complex elementary reflector 1606
complex vector
1-norm using true absolute value 1518
complex vector conjugation 1513
condition number estimation
p?gecon 1270
p?pocon 1272
p?trcon 1275
copying matrices
p?gemr2d 1746
p?trmr2d 1748
driver routines
p?dbsv 1437
p?dtsv 1440
p?gbsv 1435
p?gels 1453
p?gsvd 1439
p?gesv 1518
p?gesvx 1526
p?heev 1472
p?heevx 1475
p?heevd 1475
p?heevx 1478
p?heev 1482
p?heevx 1489
p?heevd 1492
p?heevx 1495
p?heev 1499
p?pbsv 1502
p?posv 1505
p?posvx 1507
p?ptsv 1509
nonsymmetric eigenvalue problems
p?gehrd 1510
p?lahqr 1512
p?ormh 1513
p?unmhr 1515
QR factorization
p?geqrf 1516
p?geqr2 1518
p?ggqrf 1520
p?orgqr 1522
p?ormqr 1524
QR factorization
p?gelqf 1526
p?gelq2 1528
p?ggrqf 1529
p?orgrq 1530
p?ormrq 1532
RQ factorization
p?gerqf 1534
p?gerq2 1536
p?orgqr 1537
p?ormqr 1539
RZ factorization
p?ormrz 1540
p?tzrzf 1542
singular value decomposition
p?gebrd 1543
p?orgbr 1544
p?unmbr 1546
solution refinement and error estimation
p?gerfs 1277
p?poets 1281
solving linear equations
p?dtrsv 1728
p?ptrsv 1729
symmetric eigenproblems
p?getrs 1257
p?gtrsv 1260
p?gbtrs 1259
p?gbtrs 1262
p?pots 1265
p?pots 1264
p?trtrs 1268
utility functions and routines (continued)
LQ factorization (continued)
p?ormlq 1316
p?unmql 1314
p?unmlq 1318
LU factorization
p?dtrsv 1519
p?ddtrsv 1521
p?dtrsv 1523
matrix equilibration
p?geequ 1292
p?poequ 1294
matrix inversion
p?getri 1287
p?pots 1289
p?trtri 1291
nonsymmetric eigenvalue problems
p?gehrd 1398
p?lahqr 1407
p?ormh 1409
p?unmhr 1404
Q factorization
p?geqrf 1316
p?geqr2 1318
p?ggqrf 1319
p?orgqr 1321
p?ormqr 1323
RQ factorization
p?gelqf 1325
p?gelq2 1327
p?unmql 1329
RZ factorization
p?ormrz 1332
p?tzrzf 1334
singular value decomposition
p?gebrd 1336
p?orgbr 1338
p?unmbr 1340
solution refinement and error estimation
p?gerfs 1277
p?poets 1281
solving linear equations
p?dtrsv 1728
p?ptrsv 1729
symmetric eigenproblems
p?getrs 1257
p?gtrsv 1260
p?gbtrs 1259
p?gbtrs 1262
p?pots 1265
p?pots 1264
p?trtrs 1268
utility functions and routines (continued)
symmetric eigenproblems (continued)
p?hetrd 1383
p?ormtr 1374
p?stebr 1389
p?stein 1394
p?syrtd 1371
p?unmtr 1386
symmetric eigenvalue problems
?stein2 1722
?steqr2 1731
trapezoidal matrix 1628
triangular factorization
?dbtrf 1725
?dttrf 1727
?getrf 1239
?p?gbtrf 1324
?p?potrf 1247
singular value decomposition
LAPACK 777
LAPACK routines, singular value decomposition 1412
ScaLAPACK 1412, 1489
See also LAPACK routines, singular value decomposition 777
Singular Value Decomposition 1006
triangular system of equations 1626
updating sum of squares
nabla 1618
utility functions and routines
desnict 1744
numroc 1745
?p?labad 1741
?p?lachkieee 1742
?p?lamch 1743
?p?lasnbt 1744
pxerbla 2406
triangular matrix 119
scaling
general rectangular matrix 1569
symmetric/Hermitian matrix 1584
scaling factors
general rectangular distributed matrix 1292
Hermitian positive definite distributed matrix 1294
symmetric positive definite distributed matrix 1294
scattering compressed sparse vector’s elements into full storage form 130
Schur decomposition 901, 902, 1570
Schur factorization
reordering 1676
scsum1 1105
second/dsecond 2410
Service Functions 1896
Service Routines 2037
SetInternalDecimation 2127
sgbcon 501
sgbfsx 533
sgbsvx 635
sgbtrs 461
sgecon 499
sgegs 1011
sgees 731
sgees 731
sgtrsf 539
shgeqz 893
shseqr 859
simple driver 1237
Single Dynamic Library
mkl_set_exit_handler 2408
mkl_set_interface_layer 2422
mkl_set_pardiso_pivot 2425
mkl_set_progress 2424
mkl_set_threading_layer 2423
mkl_set_xerbla 2424
single node matrix 1683
size
mkl_get_size_compact 268
sjacobi 2387
sjacobi_delete 2386
sjacobi_init 2384
sjacobi_solve 2385
sjacobi 2388
SkipAheadStream 2054
SkipAheadStreamEx 2056
slapmr 1120
slapmt 1122
slapst 1684
slaqr6 1685
slar1va 1688
slaref 1689
slarfb 1124
slarft 1129
slarfb 1692
slarbd 1694
slarre 1697
slarre 1697
slarre 1701
slarre 1704
slarrv 1706
slartg 1140
slartgs 1141
slausm 1193
small subdiagonal element 1616
smallest absolute value of a vector element 57
SNewAbstractStream 2044
solver
direct 2521
iterative 2521
Solver
Sparse 1752
solving linear equations 461
solving linear equations. linear equations 1255
sorbb 931
sorcst 1032
sorcsd 2by1 1035
sorting
eigenpairs 1710
numbers in increasing/decreasing order
LAPACK 1168
ScaLAPACK 1711
Sparse BLAS Level 1
data types 123
calling conventions 123
Sparse BLAS Level 1 routines and functions
?axpy 124
?doti 126
?doti 125
Sparse BLAS Level 1 routines and functions (continued)
?qdotui 127
?qgthr 127
?qgthr 128
?qroti 129
?qsctr 130
Sparse BLAS Level 2
naming conventions 131
sparse BLAS Level 2 routines
mkl_?bsrgemv 142
mkl_?bsrmv 175
mkl_?bsrsv 183
mkl_?bsrsvy 148
mkl_?bsrsvr 154
mkl_?coogemv 144
mkl_?coomv 179
mkl_?coosv 187
mkl_?coosymv 150
mkl_?cootrsv 156
mkl_?csclmv 177
mkl_?cscsv 185
mkl_?cssrgemv 141
mkl_?cssrsv 173
mkl_?cssrsv 181
mkl_?cssrsymv 147
mkl_?csrtrsv 152
mkl_?diagemv 145
mkl_?diamv 207
mkl_?diaav 211
mkl_?diasyymv 151
mkl_?dietrsv 158
mkl_?skymv 209
mkl_?skyv 213
Sparse BLAS Level 3
naming conventions 131
sparse BLAS Level 3 routines
mkl_?bsrrmm 191
mkl_?bsrsm 205
mkl_?coomm 196
mkl_?coosm 203
mkl_?cscmm 194
mkl_?cscsm 201
mkl_?csradd 235
mkl_?csrmm 189
mkl_?cscsymv 189
mkl_?cscsymv 159
mkl_?cscsynv 164
mkl_?cssradd 186
Sparse BLAS routines
mkl_?cbsrrsr 227
mkl_?cbsrsv 225
mkl_?cssrsc 229
mkl_?csrdia 231
mkl_?cscsymv 233
mkl_?dscarr 222
sparse matrices 130
sparse matrix 130
Sparse Matrix Checker Routines 1865
Sparse Matrix Storage Formats 132
sparse solver
parallel direct sparse solver interface
mkl_pardiso_pivot 1768
pardiso 1759
pardiso 64 1766
pardiso_getdiag 1769
pardiso_getenv 1767
pardiso_handle_delete 1774
pardiso_handle_delete 64 1776
pardiso_handle_restore 1773
pardiso_handle_restore 64 1775
pardiso_handle_store 1772
pardiso_handle_store 64 1774
pardiso_setenv 1767
pardisoinit 1765
parallel direct sparse solver interface for clusters
cluster_sparse_solver 1796
cluster_sparse_solver 64 1801
Sparse Solver
direct sparse solver interface
dss_create 1821
dss_define_structure
dss_define_structure 1822
dss_delete 1829
dss_factor 1825
dss_factor_complex 1825
dss_factor_real 1825
dss_reorder 1824
dss_solve 1827
dss_solve_complex 1827
dss_solve_real 1827
dss_statistics 1829
iterative sparse solver interface
dcg 1846
dcg_check 1845
dcg_get 1847
dcg_init 1844
dcgmrhs 1850
dcgmrhs_check 1849
dcgmrhs_get 1852
dcgmrhs_init 1848
dfgmres 1854
dfgmres_check 1853
dfgmres_get 1856
dfgmres_init 1852
preconditioners based on incomplete LU factorization
dcsfiru0 1859
dcsrlit 1862
Sparse Solvers 1752, 1770, 1777, 1781, 1794, 1809, 1869, 1870, 1872, 1876, 1881
sparse vectors
adding and scaling 124
complex dot product, conjugated 126
complex dot product, unconjugated 127
compressed form 123
converting to compressed form 127, 128
converting to full-storage form 130
full-storage form 123
Givens rotation 129
norm 123
passed to BLAS level 1 routines 123
real dot product 125
scaling 123
sparse_matrix_checker 1865
sparse_matrix_checker_init
sparse_request_t
sparse_stage_finalize_mult
sparse_stage_full_mult
two-stage algorithm
sparse_stage_nnz_count
specific hardware support
  mkl_enable_instructions
  mkl_finalize
  mkl_set_mpi
Spline Methods
  split Cholesky factorization (band matrices)
  spsrf
  spprfs
  spptrs
  square matrix
    1-norm estimation
    LAPACK
    ScALAPACK
ssteg2 1712
ssteg2a 1715
ssteg2b 1719
ssycon_3 511
ssyconv 1101
ssyev_a 683
ssyev_rk 686
ssywaper 1194
ssytrf_a 440
ssytrf_rk 443
ssytria 593
ssytri2 588
ssytrix 590
ssytrs_3 487
ssytris 484
stream 2034
strexc 877
stride. increment 2539
strmv 1732
strnlsp_check 2372
strnlsp_delete 2376
strnlsp_get 2375
strnlsp_init 2370
strnlsp_solve 2373
strnlspbc_check 2379
strnlspbc_delete 2383
strnlspbc_get 2382
strnlspbc_init 2377
strnlspbc_solve 2381
sum
  of distributed vectors 2248
  of magnitudes of elements of a distributed vector 2247
  of magnitudes of the vector elements 45
  of sparse vector and full-storage vector 124
  of vectors 46, 349
sum of squares
  updating
  ScALAPACK
  summary statistics
summary statistics (continued)
  vsldsseditmoments 2161
  vsldSSEditMoments 2161
  vsldsseditoutliersdetection 2173
  vsldSSEditOutliersDetection 2173
  vsldSSEditPartialCovCor 2166
  vsldsseditpooledcovariance 2170
  vsldSSEditPooledCovariance 2170
  vsldsseditquantiles 2168
  vsldSSEditQuantiles 2168
  vsldsseditstreamquantiles 2169
  vsldSSEditStreamQuantiles 2169
  vsldsseditsums 2162
  vsldSSEditSums 2162
  vsldssedittask 2152
  vsldSSEditTask 2152
  vsldssnewtask 2149
  vsldSSEditNewTask 2149
  vsslssedittask 2152
  vsslSSeditTask 2152
  vsslssdeletetask 2183
  vsslSSEditDeleteTask 2183
  vsslsscompute 2182
  vsslSSCompute 2182
  vsslsseditcorparameterization 2178
  vsslSSEditCorParameterization 2178
  vsslsseditcovcor 2163
  vsslSSEditCovCor 2163
  vsslsseditpartialcovariance 2170
  vsslSSEditPartialCovCor 2170
  vsslsseditpooledcovariance 2170
  vsslSSEditPooledCovariance 2170
  vsslsseditquantiles 2168
  vsslSSEditQuantiles 2168
  vsslsseditstreamquantiles 2169
  vsslSSEditStreamQuantiles 2169
  vsslsseditsums 2161
  vsslSSEditSums 2161
  vsslsseditoutliersdetection 2173
  vsslSSEditOutliersDetection 2173
  vsslsseditpartialcovariance 2170
  vsslSSEditPartialCovCor 2170
  vsslsseditpooledcovariance 2170
  vsslSSEditPooledCovariance 2170
  vsslsseditquantiles 2168
  vsslSSEditQuantiles 2168
  Summary Statistics
  summary statistics usage examples
  support functions
    mkl_calloc 2417
    mkl_enable_instructions 2433
    mkl_finalize 2439
    mkl_free 2419
    mkl_malloc 2417
    mkl_realloc 2418
    mkl_set_env_mode 2435
    mkl_set_mpi 2437
    mkl_verbose 2436
support functions (continued)
mkl.verbose.output_file 2437

Support Functions
exception handling 2404
handling fatal errors 2408

support functions for CNR
mkl.cbwr.get 2427
mkl.cbwr.get_auto_branch 2428
mkl.cbwr.set 2427

support routines
mkl.disable.fast_mm 2414
mkl.freeBuffers 2413
mkl.set.memory_limit 2419
mkl.thread.freeBuffers 2414

progress information 2432

SVD (singular value decomposition)
LAPACK 777
ScalAPACK 1412

swapping distributed vectors 2256
swapping vectors 55
Sylvester’s equation 882
symmetric distributed matrix
rank-n update 2304, 2306
scalar-matrix-matrix product 2301

Symmetric Eigenproblems 947
symmetric matrix
Bunch-Kaufman factorization
packed storage 454
eigenvalues and eigenvectors 1456, 1459, 1462, 1466
estimating the condition number
packed storage 516
generalized eigenvalue problems 834
inverting the matrix
packed storage 596
matrix-vector product
band storage 78
packed storage 81
packed storage 458
rank-1 update
packed storage 82
rank-2 update
packed storage 84
rank-2k update 116
rank-n update 114
reducing to standard form
ScalAPACK 1666
reducing to tridiagonal form
ScalAPACK 1623
scalar-matrix-matrix product 112
scaling 1584
solving systems of linear equations
packed storage 490
symmetric positive definite distributed matrix
computing scaling factors 1294
equilibration 1294
symmetric positive semidefinite matrix
Cholesky factorization 430
symmetric positive-definite distributed matrix
inverting the matrix 1289
symmetric positive-definite matrix
Cholesky factorization
band storage 434, 1249
packed storage 433
ScalAPACK 1247, 1661
estimating the condition number
band storage 507
packed storage 506
tridiagonal matrix 508

solving systems of linear equations (continued)
inverting the matrix
packed storage 584
solving systems of linear equations
band storage 472, 1263
LAPACK 467
packed storage 470
ScalAPACK 1262

symmetric positive-definite tridiagonal matrix
solving systems of linear equations 1265
symmetric tridiagonal matrix
eigenvalues and eigenvectors 1694
system of linear equations
with a distributed triangular matrix 2287
with a triangular matrix
band storage 92
packed storage 97

systems of linear equations
linear equations 1728
systems of linear equations linear equations 1253
sympart 1194
sytri2 588
sytri2x 590

T

Task Computation Routines 2499
Task Creation and Initialization 2482
Task Status 2480

threading control
mkl.domain.get_max_threads 2401
mkl.domain.set_num_threads 2397
mkl.get_dynamic 2402
mkl.get_max_threads 2400
mkl.get_num_stripes 2404
mkl.set_dynamic 2399
mkl.set_num_stripes 2403
mkl.set_num_threads 2396
mkl.set_num_threads_local 2398

Threading Control 2395

timing functions
mkl.get_clocks_frequency 2412
MKL_Get_Cpu_Clocks 2411
mkl.get_cpu_frequency 2411
mkl.get_max_cpu_frequency 2412
second/dsecnd 2410

TR routines
?tnlsps_check 2372
?tnlsps_delete 2376
?tnlsps_get 2375
?tnlsps_init 2370
?tnlsps_solve 2373
?tnlspsbpc_check 2379
?tnlspsbpc_delete 2383
?tnlspsbpc_get 2382
?tnlspsbpc_init 2377
?tnlspsbpc.solve 2381

nonlinear least squares problem
with linear bound constraints 2377
without constraints 2370
organization and implementation 2368

transposition
distributed complex matrix 2309
distributed complex matrix, conjugated 2310
distributed real matrix 2308
Transposition and General Memory Movement Routines 348

transposition parameter 2542

trapezoidal matrix
RZ factorization (continued)
1-norm value 1118
copying 1748
Frobenius norm 1118
infinity- norm 1118
largest absolute value of element 1118
reduction to triangular form 1628
RZ factorization
LAPACK 762
ScalAPACK 1349
trex 877
triangular banded equations
ScalAPACK 1653
triangular distributed matrix
inverting the matrix 1291
scalar-matrix-matrix product 2312
triangular factorization
band matrix 421, 1241, 1243, 1519, 1725
diagonally dominant tridiagonal matrix
LAPACK 425
general matrix 414, 416, 1239
Hermitian matrix
packed storage 456
Hermitian positive semidefinite matrix 430
Hermitian positive-definite matrix
band storage 434, 1249
packed storage 433
tridiagonal matrix 436, 1251
symmetric matrix
packed storage 454, 458
symmetric positive semidefinite matrix 430
symmetric positive-definite matrix
band storage 434, 1249
packed storage 433
tridiagonal matrix 436, 1251
tridiagonal matrix
LAPACK 424
ScalAPACK 1727
triangular matrix
1-norm value
LAPACK 1118
ScalAPACK 1563
copying 1202, 1203, 1210–1213
estimating the condition number
band storage 521
packed storage 520
Frobenius norm
LAPACK 1118
ScalAPACK 1563
infinity- norm
LAPACK 1118
ScalAPACK 1563
inverting the matrix
packed storage 601
ScalAPACK 1680
largest absolute value of element
LAPACK 1118
ScalAPACK 1563
matrix-vector product
band storage 89
packed storage 95
product
blocked algorithm 1193, 1631
LAPACK 1193
ScalAPACK 1630, 1631
ScalAPACK 1398
scalar-matrix-matrix product 119
solving systems of linear equations
solving systems of linear equations (continued)
solving systems of linear equations (continued)
band storage 92, 497
packed storage 97, 495
ScalAPACK 1268
triangular matrix factorization
Hermitian positive-definite matrix 426
symmetric positive-definite matrix 426
triangular pentagonal matrix
QR factorization 772, 1205, 1207
triangular system of equations
solving with scale factor
ScalAPACK 1626
tridiagonal matrix
base representations and eigenvalues 1697, 1701
eigenvalues and eigenvectors 1712, 1719
eigenvectors 1706
estimating the condition number 503
solving systems of linear equations
ScalAPACK 1728
tridiagonal triangular factorization
band matrix 1521
tridiagonal triangular system of equations 1659
trigonometric transform
backward cosine 2318
backward sine 2318
backward staggered cosine 2318
backward staggered sine 2318
backward twice staggered cosine 2318
backward twice staggered sine 2318
forward cosine 2318
forward sine 2318
forward staggered cosine 2318
forward staggered sine 2318
forward twice staggered cosine 2318
forward twice staggered sine 2318
Trigonometric Transform interface
routines
?_backward_trig_transform 2326
?_commit_trig_transform 2322
?_forward_trig_transform 2324
?_init_trig_transform 2321
free_trig_transform 2328
Trigonometric Transforms interface 2321
TT interface 2317
TT routines 2321
two matrices
QR factorization
LAPACK 767
ScalAPACK 1357

U
ungbr 787
Uniform (continuous) 2062
Uniform (discrete) 2087
UniformBits 2089
UniformBits32 2090
UniformBits64 2091
unitary matrix
CS decomposition
LAPACK 928, 931, 1031, 1035
from LQ factorization
ScalAPACK 1638
from QL factorization
ScalAPACK 1634, 1642
from QR factorization
ScalAPACK 1636
from RQ factorization (continued)
from RQ factorization
ScalAPACK 1640
ScalAPACK 1398, 1412
Unpack Functions 1896
updati
rank-1
distributed general matrix 2263
distributed Hermitian matrix 2272
distributed symmetric matrix 2279
gen eral matrix 63
Hermitian matrix 71, 75
real symmetric matrix 82, 87
rank-1, conjugated
distributed general matrix 2265
gen eral matrix 64
rank-1, unconjugated
distributed general matrix 2267
gen eral matrix 66
rank-2
distributed Hermitian matrix 2274
distributed symmetric matrix 2281
Hermitian matrix 72, 77
symmetric matrix 84, 88
rank-2k
Hermitian distributed matrix 2299
Hermitian matrix 109, 257, 260, 262
symmetric distributed matrix 2306
symmetric matrix 116
rank-k
distributed Hermitian matrix 2297
Hermitian matrix 107
symmetric distributed matrix 2304
rank-n
symmetric matrix 114
updati:rank-1
Hermitian matrix
packed storage 75
real symmetric matrix
packed storage 82
updati:rank-2
Hermitian matrix
packed storage 77
symmetric matrix
packed storage 84
upper Hessenberg matrix
1-norm value
ScalAPACK 1559
Frobenius norm
ScalAPACK 1559
infinity- norm
ScalAPACK 1559
largest absolute value of element
ScalAPACK 1559
ScalAPACK 1398

V
v?Abs 1909
v?Acos 1952
v?Acosh 1978
v?Add 1901
v?Arg 1910
v?Asin 1954
v?Asinh 1979
v?Atan 1956
v?Atan2 1957
v?Atanh 1981
v?Cbrt 1921
v?CdfNorm 1987
v?CdfNormInv 1992
v?CIS 1950
v?Conj 1908
v?Cos 1945
v?Cosh 1972
v?Div 1917
v?Erf 1983
v?Erfc 1985
v?ErfcInv 1991
v?ErfInv 1988
v?Exp 1932
v?ExpInt 1996
v?Expm1 1937
v?Floor 1997
v?Frac 2004
v?Hypot 1931
v?Inv 1916
v?InvCbrt 1922
v?InvSqrt 1920
v?Igamma 1994
v?LGamma 1994
v?LinearFrac 1911
v?Ln 1938
v?Log 1941
v?Log1p 1943
v?Modf 2003
v?Mul 1905
v?MulByConj 1907
v?NearlyInt 2001
v?Pack 2006
v?Pow 1925
v?Pow2o3 1923
v?Pow3o2 1924
v?Powx 1928
v?Rint 2002
v?Round 2000
v?Sin 1947
v?SinCos 1948
v?Sinh 1974
v?Sqrt 1904
v?Sqr 1919
v?Sub 1903
v?Tan 1951
v?Tanh 1976
v?Tgamma 1995
v?TGamma 1995
v?Trunc 1999
v?Unpack 2007
cvAdd 1901
cvPackl 2006
cvPackM 2006
cvPackV 2006
cvSin 1947
cvSub 1903
cvUnpackl 2007
cvUnpackM 2007
cvUnpackV 2007
dvAcospi 1963
dvAdd 1901
dvAsinpi 1964
dvAtan2pi 1966
dvAtanpi 1965
dvCopySign 2016
dvCosd 1968
dvCosp 1959
vector mathematical functions (continued)
  inverse cumulative normal distribution function 1992
  inverse error function value 1988
  inverse hyperbolic cosine 1978
  inverse hyperbolic sine 1979
  inverse hyperbolic tangent 1981
  inverse sine 1954
  inverse square root 1920
  inverse tangent 1956
  inversion 1916
  linear fraction transformation 1911
  multiplication 1905
  multiplication of conjugated vector element 1907
  natural logarithm 1938
  natural logarithm of vector elements increased by 1 1943
  power 1925
  power (constant) 1928
  power 2/3 1923
  power 3/2 1924
  rounding to nearest integer value 2000
  rounding towards minus infinity 1997, 2004
  rounding towards plus infinity 1998
  rounding towards zero 1999
  sine 1947
  sine and cosine 1948
  square root 1919
  square root of sum of squares 1931
  squaring 1904
  subtraction 1903
  tangent 1951

Vector Mathematical Functions 1894

vector multiplication 2006

vector statistics functions
  Bernoulli 2093
  Beta 2084
  Binomial 2096
  Cauchy 2074
  CopyStream 2045
  CopyStreamState 2046
  DeleteStream 2045
  dNewAbstractStream 2042
  Exponential 2069
  Gamma 2082, 2086
  Gaussian 2064
  GaussianMV 2066
  Geometric 2094
  GetBrngProperties 2108
  GetNumRegBrngs 2058
  GetStreamSize 2051
  GetStreamStateBrng 2057
  Gumbel 2080
  Hypergeometric 2097
  iNewAbstractStream 2041
  Laplace 2071
  LeapfrogStream 2052
  LoadStreamF 2048
  LoadStreamM 2050
  Lognormal 2078
  Multinomial 2104
  NegBinomial 2103
  NewStream 2038
  NewStreamEx 2040
  Poisson 2099
  PoissonV 2101
vector statistics functions (continued)
- Rayleigh 2076
- RegisterBrng 2107
- SaveStreamF 2047
- SaveStreamM 2049
- SkipAheadStream 2054
- SkipAheadStreamEx 2056
- sNewAbstractStream 2044
- Uniform (continuous) 2062
- Uniform (discrete) 2087
- UniformBits 2089
- UniformBits32 2090
- UniformBits64 2091
- Weibull 2073

vector unpack function 2007
vector-scalar product
- sparse vectors 124

vectors
- adding magnitudes of vector elements 45
- copying 46
  - dot product
  - complex vectors 49
  - complex vectors, conjugated 49
  - real vectors 47
- element with the largest absolute value 56
- element with the largest absolute value of real part and
  its index 1517
- element with the smallest absolute value 57
- Euclidean norm 50
- Givens rotation 52
- linear combination of vectors 46, 349
- modified Givens transformation parameters 53
- rotation of points 51
- rotation of points in the modified plane 52
- sparse vectors 123
- sum of vectors 46, 349
- swapping 55
- vector-scalar product 55

viRngUniformBits 2089
viRngUniformBits32 2090
viRngUniformBits64 2091

VM
- Functions Interface 1896
  - Input Parameters 1896
  - Output Parameters 1897
- VM arithmetic functions 1901
- VM exponential and logarithmic functions 1932
- VM functions
  - mathematical functions
    - v?Abs 1909
    - v?Acos 1952
    - v?Acosh 1978
    - v?Add 1901
    - v?Arg 1910
    - v?Asin 1954
    - v?Asinh 1979
    - v?Atan 1956
    - v?Atan2 1957
    - v?Atanh 1981
    - v?Cbrt 1921
    - v?CdfNorm 1987
    - v?CdfNormInv 1992
    - v?CIS 1950
    - v?Conj 1908
    - v?Cos 1945
    - v?Cosh 1972
    - v?Div 1917
  - service functions (continued)
    - mathematical functions (continued)
      - v?Erf 1983
      - v?Erfc 1985
      - v?ErfcInv 1991
      - v?ErfInv 1988
      - v?Exp 1932
      - v?ExpInt1 1996
      - v?Expm1 1937
      - v?Floor 1997
      - v?Frac 2004
      - v?Hypot 1931
      - v?Inv 1916
      - v?InvCbrt 1922
      - v?InvSqrt 1920
      - v?LGamma 1994
      - v?LinearFrac 1911
      - v?Ln 1938
      - v?Log10 1941
      - v?Log1p 1943
      - v?Modf 2003
      - v?Mul 1905
      - v?MulByConj 1907
      - v?NearbyInt 2001
      - v?Pow 1925
      - v?Pow2o3 1923
      - v?Pow3o2 1924
      - v?Powx 1928
      - v?Rint 2002
      - v?Round 2000
      - v?Sin 1947
      - v?SinCos 1948
      - v?Sinh 1974
      - v?Sqr 1904
      - v?Sqrt 1919
      - v?Sub 1903
      - v?Tan 1951
      - v?Tanh 1976
      - v?TGamma 1995
      - v?Trunc 1999
  - pack/unpack functions
    - v?Pack 2006
    - v?Unpack 2007
  - service functions
    -_ClearErrorCallBack 2016
    - _ClearErrorStatus 2013
    - _GetErrorCallBack 2016
    - _GetErrStatus 2013
    - _GetMode 2010
    - _MKLFreeTls 2011
    - _setError 2014
    - _SetErrStatus 2012
    - _SetMode 2009
- VM hyperbolic functions 1972
- VM mathematical functions
  - arithmetic 1901
  - exponential and logarithmic 1932
  - hyperbolic 1972
  - miscellaneous 2016
  - power and root 1916
  - rounding 1997
  - special 1983
  - special value notations 1901
  - trigonometric 1945
- VM Mathematical Functions 1896
- VM Miscellaneous Functions 2016
- VM Pack Functions 1896
- VM Pack/Unpack Functions 2005
<table>
<thead>
<tr>
<th>Function</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM power and root functions</td>
<td>1916</td>
</tr>
<tr>
<td>VM rounding functions</td>
<td>1997</td>
</tr>
<tr>
<td>VM Service Functions</td>
<td>2008</td>
</tr>
<tr>
<td>VM special functions</td>
<td>1983</td>
</tr>
<tr>
<td>VM trigonometric functions</td>
<td>1945</td>
</tr>
<tr>
<td>vmcAdd</td>
<td>1901</td>
</tr>
<tr>
<td>vmcSin</td>
<td>1947</td>
</tr>
<tr>
<td>vmcSub</td>
<td>1903</td>
</tr>
<tr>
<td>vmdAcospi</td>
<td>1963</td>
</tr>
<tr>
<td>vmdAsinpi</td>
<td>1964</td>
</tr>
<tr>
<td>vmdAtan2pi</td>
<td>1966</td>
</tr>
<tr>
<td>vmdAtanpi</td>
<td>1965</td>
</tr>
<tr>
<td>vmdCopySign</td>
<td>2016</td>
</tr>
<tr>
<td>vmdCosd</td>
<td>1968</td>
</tr>
<tr>
<td>vmdCospi</td>
<td>1959</td>
</tr>
<tr>
<td>vmdExp10</td>
<td>1936</td>
</tr>
<tr>
<td>vmdExp2</td>
<td>1934</td>
</tr>
<tr>
<td>vmdFdim</td>
<td>2018</td>
</tr>
<tr>
<td>vmdFmax</td>
<td>2019</td>
</tr>
<tr>
<td>vmdFmin</td>
<td>2020</td>
</tr>
<tr>
<td>vmdFmod</td>
<td>1914</td>
</tr>
<tr>
<td>vmdLog2</td>
<td>1940</td>
</tr>
<tr>
<td>vmdLogb</td>
<td>1944</td>
</tr>
<tr>
<td>vmdMaxMag</td>
<td>2022</td>
</tr>
<tr>
<td>vmdMinMag</td>
<td>2023</td>
</tr>
<tr>
<td>vmdNextAfter</td>
<td>2017</td>
</tr>
<tr>
<td>vmdPowr</td>
<td>1930</td>
</tr>
<tr>
<td>vmdRemainder</td>
<td>1915</td>
</tr>
<tr>
<td>vmdSin</td>
<td>1947</td>
</tr>
<tr>
<td>vmdSind</td>
<td>1969</td>
</tr>
<tr>
<td>vmdSinpi</td>
<td>1960</td>
</tr>
<tr>
<td>vmdSub</td>
<td>1903</td>
</tr>
<tr>
<td>vmdTand</td>
<td>1971</td>
</tr>
<tr>
<td>vmdTanpi</td>
<td>1961</td>
</tr>
<tr>
<td>vmlClearErrorCallBack</td>
<td>2016</td>
</tr>
<tr>
<td>vmlClearErrStatus</td>
<td>2013</td>
</tr>
<tr>
<td>vmlGetErrorCallBack</td>
<td>2016</td>
</tr>
<tr>
<td>vmlGetErrStatus</td>
<td>2013</td>
</tr>
<tr>
<td>vmlGetMode</td>
<td>2010</td>
</tr>
<tr>
<td>vmlSetErrorCallBack</td>
<td>2014</td>
</tr>
<tr>
<td>vmlSetErrStatus</td>
<td>2012</td>
</tr>
<tr>
<td>vmlSetMode</td>
<td>2009</td>
</tr>
<tr>
<td>vmsAcospi</td>
<td>1963</td>
</tr>
<tr>
<td>vmsAdd</td>
<td>1901</td>
</tr>
<tr>
<td>vmsAsinpi</td>
<td>1964</td>
</tr>
<tr>
<td>vmsAtan2pi</td>
<td>1966</td>
</tr>
<tr>
<td>vmsAtanpi</td>
<td>1965</td>
</tr>
<tr>
<td>vmsCopySign</td>
<td>2016</td>
</tr>
<tr>
<td>vmsCosd</td>
<td>1968</td>
</tr>
<tr>
<td>vmsCospi</td>
<td>1959</td>
</tr>
<tr>
<td>vmsExp10</td>
<td>1936</td>
</tr>
<tr>
<td>vmsExp2</td>
<td>1934</td>
</tr>
<tr>
<td>vmsFdim</td>
<td>2018</td>
</tr>
<tr>
<td>vmsFmax</td>
<td>2019</td>
</tr>
<tr>
<td>vmsFmin</td>
<td>2020</td>
</tr>
<tr>
<td>vmsFmod</td>
<td>1914</td>
</tr>
<tr>
<td>vmsLog2</td>
<td>1940</td>
</tr>
<tr>
<td>vmsLogb</td>
<td>1944</td>
</tr>
<tr>
<td>vmsMaxMag</td>
<td>2022</td>
</tr>
<tr>
<td>vmsMinMag</td>
<td>2023</td>
</tr>
<tr>
<td>vmsNextAfter</td>
<td>2017</td>
</tr>
<tr>
<td>vmsPowr</td>
<td>1930</td>
</tr>
<tr>
<td>vmsRemainder</td>
<td>1915</td>
</tr>
<tr>
<td>vmsSin</td>
<td>1947</td>
</tr>
<tr>
<td>vmsSind</td>
<td>1969</td>
</tr>
<tr>
<td>vmsSinpi</td>
<td>1960</td>
</tr>
<tr>
<td>vmsSub</td>
<td>1903</td>
</tr>
<tr>
<td>vmsTand</td>
<td>1971</td>
</tr>
<tr>
<td>vmsTanpi</td>
<td>1961</td>
</tr>
<tr>
<td>vmzAdd</td>
<td>1901</td>
</tr>
<tr>
<td>vmzSin</td>
<td>1947</td>
</tr>
<tr>
<td>vmzSub</td>
<td>1903</td>
</tr>
</tbody>
</table>

**VM routines**
- advanced service routines
- convolution/correlation
- CopyTask
- DeleteTask
- Exec
- Exec1D
- ExecX
- GetBrngProperties
- RegisterBrng
- CopyStream
- CopyStreamState
- DeleteStream
- dNewAbstractStream
- GetNumRegBrngs
- GetStreamSize
- GetStreamStateBrng
- iNewAbstractStream
- LeapfrogStream
- LoadStream
- SaveStream
- SkipAheadStream
- summary statistics
- Compute
- DeleteTask
summary statistics (continued)
EditCorParameterization 2178
EditCovCor 2163
EditCP 2165
EditMissingValues 2174
EditMoments 2161
EditOutliersDetection 2173
EditPartialCovCor 2166
EditPooledCovCor 2160
EditRobustCovariance 2170
EditQuantiles 2168
EditStreamQuantiles 2169
EditSums 2162
EditTask 2152
NewTask 2149
VS routines: convolution/correlation
SetInternalDecimation 2127
SetMode 2124
SetStart 2126
VS task 2025
vsAcospi 1963
vsAdd 1901
vsAsinpi 1964
vsAtan2pi 1966
vsAtanpi 1965
vsCopySign 2016
vsCosd 1968
vsCospi 1959
vsExp10 1936
vsExp2 1934
vsFdim 2018
vsFmax 2019
vsFmin 2020
vsFmod 2021
vslConvCopyTask 2136
vslCorrCopyTask 2136
vsldsscompute 2182
vsldSSCompute 2182
vsldsseditcorparameterization 2178
vslldSSEditCorParameterization 2178
vslldSSEditCovCor 2163
vslldSSEditCovCor 2163
vslldsseditcp 2165
vslldSSEditCP 2165
vslldsseditmissingvalues 2174
vslldSSEditMissingValues 2174
vslldsseditmoments 2161
vslldSSEditMoments 2161
vslldsseditoutliersdetection 2173
vslldSSEditOutliersDetection 2173
vslldSSEditPartialCovCor 2166
vslldSSEditPooledCovariance 2170
vslldSSEditRobustCovariance 2170
vslldSSEditQuantiles 2168
vslldSSEditSums 2162
vslldSSEditTask 2152
vslldSSEditTask 2152
vslldssnewtask 2149
vslldSSNewTask 2149
vslggamma 1994
vslLGamma 1994
W
Weibull 2073
Wilkinson transform 1633

X
xerbla 2404
xerbla, error reporting routine 41, 1897, 2243

Z
zbbcsd 928
zgbcn 501
zgbrrfsx 533
zgbsvx 635
zgbrtrs 461
zgecon 499
zgeqpf 731
zgttrs 539
zhecon_3 514
zhesv_aa 699
zhesv_rk 701
zheswapr 1196
zhetrf_aa 448
zhetrf_rk 451
zhetri_3 594
zhetri2 589
zhetri2x 591
zhetrs_3 488
zhetrs_aa 481
zhetrs2 485
zhgeqz 893
zhseqr 859
zlahqr2 1681
zlapmr 1120
zlapmt 1122
zlarf 1689
zlarfb 1124
zlarft 1129
zlaum 1193
zporfsx 543
zpprfs 549
zpptrs 470
zsycon_3 511
zsyconv 1101
zsysv_aa 683
zsysv_rk 686
zsyswapr 4194
zsytrf_aa 440
zsytrf_rk 443
zsytri_3 593
zsytri2 588
zsytri2x 590
zsysr_3 487
zsysr_aa 477
zsysr2 484
ztrexc 877
ztrmv 1732
zunbdb 931
zuncsd 1032
zuncsd2by1 1035
zungr 787